ADAPTIVE HYBRID SWITCHING TECHNIQUE FOR PARALLEL COMPUTING SYSTEM

by

Zhu Ding

Master of Science, Southeast University (Nanjing, China), 2000

Submitted to the Graduate Faculty of

the School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This dissertation was presented

by

Zhu Ding

It was defended on

March 27, 2006

and approved by

Raymond R. Hoare, Ph.D., Assistant Professor, Electrical and Computer Engineering Deptartment

James T. Cain, Ph.D., Professor, Electrical and Computer Engineering Department

Ronald G. Hoelzeman, Ph.D., Associate Professor, Electrical and Computer Engineering Department

Alex K. Jones, Ph.D., Assistant Professor, Electrical and Computer Engineering Department

Rami Melhem, Ph.D., Professor, Computer Science Department

Dissertation Director: Raymond R. Hoare, Ph.D., Assistant Professor, Electrical and Computer Engineering Deptartment

ADAPTIVE HYBRID SWITCHING TECHNIQUE FOR PARALLEL COMPUTING SYSTEM

Zhu Ding, PhD

University of Pittsburgh, 2006

Parallel processing accelerates computations by solving a single problem using multiple compute nodes interconnected by a network. The scalability of a parallel system is limited by its ability to communicate and coordinate processing. Circuit switching, packet switching and wormhole routing are dominant switching techniques. Our simulation results show that wormhole routing and circuit switching each excel under different types of traffic.

This dissertation presents a hybrid switching technique that combines wormhole routing with circuit switching in a single switch using virtual channels and time division multiplexing. The performance of this hybrid switch is significantly impacted by the efficiency of traffic scheduling and thus, this dissertation also explores the design and scalability of hardware scheduling for the hybrid switch. In particular, we introduce two schedulers for crossbar networks: a greedy scheduler and an optimal scheduler that improves upon the results provided by the greedy scheduler. For the time division multiplexing portion of the hybrid switch, this dissertation presents three allocation methods that combine wormhole switching with predictive circuit switching.

We further extend this research from crossbar networks to fat tree interconnected networks with virtual channels. The global "level-wise" scheduling algorithm is presented and improves network utilization by 30% when compared to a switch-level algorithm.

The performance of the hybrid switching is evaluated on a cycle-accurate simulation framework that is also part of this dissertation research. Our experimental results demonstrate that the hybrid switch is capable of transferring both predictable traffics and un-

predictable traffics successfully. By dynamically selecting the proper switching technique based on the type of communication traffic, the hybrid switch improves communication for most types of traffic.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This dissertation is dedicated to my parents, who give me the utmost support and encouragement.

Importantly, I thank my advisor, Professor Raymond Hoare, for his guidance and assistance. He is an excellent teacher, researcher and mentor. I am very impressed by his intelligence. I have enjoyed working with him and learning from him. I would like to thank him for providing me the great opportunity to be involved in this interesting research project.

I want to express my gratitude to Professor Rami Melhem. He provided me very insightful and knowledgeable advice to my research. I would like to thank Professor Alex Jones. Without his assistance and his powerful workstation, it would have been impossible to generate so many simulation results for this dissertation. In addition, I would like to thank Professor Tom Cain and Professor Ronald Hoelzeman who served as committee members on my Ph.D program. I appreciate their valuable time to read and provided comments on my proposal and final dissertation.

For their friendship and constructive discussions, I would like to express my special thanks to Shenchih Tung, Ying Yu, Dara Kusic, Jeff Schuster, Kshitij Gupta, Johnny Ng, Dan Li, Jiang Zhen, Shuiyi Shao and other labmates.

Finally but the most importantly, I thank my wonderful husband, Jian Sun. Besides his love, he has given me invaluable assistance and suggestions on my research. Words can not convey my appreciation. I also want to thank my two sons, Roy and William. They make me feel proud. Their love and smiles encourage me.

1.0 INTRODUCTION

Communications in parallel computing systems limit the computing performance. Parallel processing solves a single problem by tightly coordinating the efforts of multiple processors in order to perform a particular computation faster than a single processor. Usually, the computation power increases when the number of parallel processors grows. In fact, there is a point when adding an additional processor to a computational problem actually increases the total execution time as a result of the additional communication and coordination overhead. This overhead is a function of the network's performance. The performance of the network heavily impacts the achievable speedup.

Switched interconnection communication networks used to construct scalable parallel computers, as shown in Figure 1.1. All processors (PEs) are connected to switch via the network interface card (NIC). The switch represents either a single crossbar or multiple interconnected switches. The switch carries out all communications between processors.

Research in the area of switched interconnection communication networks in parallel computing systems has been primarily focused on network routing techniques. Circuit switching, packet switching, and wormhole switching are three dominant switching methods that have been used. The first generation of parallel computing systems employed either circuit switch or packet switch. The Intel iPSC/1 was packet switched, with message packets stored in their entirety and retransmitted at each intermediate node in a hypercube network. The Intel iPSC/2 and iPSC/860 [1] use circuit-switched communication that dedicates a path when two nodes need to communicate. Wormhole routing has been used in a variety of parallel systems including the Intel Paragon, Cray T3D [2], IBM Power Parallel SP series[3], and the Quadrics switch [4].

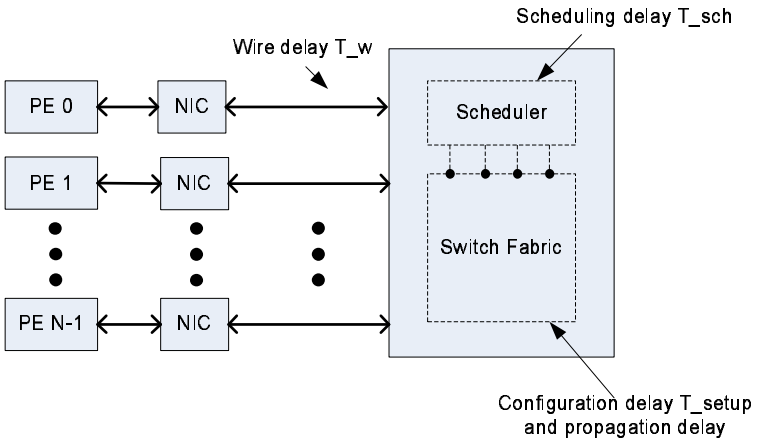


Figure 1.1: Switched interconnection network.

Recently, the development in analog switch technologies, including optical switches [5, 6] and LVDS switches [7] require that paths be established without analyzing the packet, therefore requiring circuit switching.

However, any single switching is not sufficient. The benefit of circuit switching is in the simple and low latency switch elements. More important, circuit switching does not require packet analysis and therefore is capable of encapsulating data transparent switches, such as optical switches and LVDS switches, to achieve high speed throughput. However, circuit switch has to tolerate the overhead to establish circuit connections. Wormhole switching is more feasible to transfer data packets, but it has to pay overhead caused by flow control and data buffering in each switch node. Further more, the communication within parallel computing has its own characteristics. Communication regularity means the pattern of the communications is predictable. If some communication nodes are used frequently within a given period of time, it is said that the communications have locality. Many applications running on parallel computing systems exhibit temporal and spatial locality and regularity. The network switch can benefit from the knowledge of communication characteristics.

Research has been carried out on developing more efficient switching techniques by extending current circuit switch scheme [8, 9]. However, in the previous designs, two main issues that heavily impact the network performance are not well investigated.

1. Efficiency of traffic scheduling

   The overhead of establishing connections is a major drawback for circuit switching technique. A fast scheduler will benefit both wormhole switching and circuit switching as both methods need to configure the switch elements. Moreover, in a circuit switch, the circuit connections will be held for a certain amount of time. We call this kind of connections long-lived. The less connections established, the more network bandwidth is wasted. This penalty becomes significant when it happens on long-lived connections. Therefore, a network scheduler that can establish connections fast and efficiently is crucial in switched interconnection networks.

2. Combination of circuit switch and wormhole switch

   In previous work, which is introduced in detail in Chapter 2, a certain number switches are dedicated for packet switching and a seperate number of switches are dedicated for circuit switching. In other words, the network bandwidth assigned for packet switch and circuit switch is unchangeable. However, the ratio of wormhole switched traffic and circuit switched traffic is variant in different parallel applications. There is no reason to fix the partition of network bandwidth in switch designs. Network bandwidth utilization should be improved if the network bandwidth is assigned adaptive according to communication requirement.

This dissertation presents a hybrid switching technique that combines wormhole routing with circuit switching using virtual channel in the time division multiplexing (TDM) domain. After analyzing message latency in parallel computing networks, we have found that the network scheduler's performance is important to reduce message latency. Therefore, we focus on the design and implementation of schedulers for the hybrid switch. In particular, we introduce a greedy and a optimizing scheduler for crossbar networks. In addition, we propose three schemes to allocate network bandwidth for wormhole switching and predictive circuit switching using TDM. Furthermore, we extend the crossbar networks to fat-tree

interconnected networks. The virtual channel assignment schemes is applicable, therefore, we put our effort to accomplish a level-wise scheduling algorithm for fat tree interconnection networks. The performance of the hybrid switching is evaluated based on a cycle-accurate simulation framework that is also part of this dissertation research.

In order to incorporate the collective impact of latency, peak bandwidth and contention, we show our result in terms of the effective bandwidth utilization that is calculated by dividing the total number of data bits sent by the total time required for a set of messages. We normalize this value by dividing it by peak bandwidth.

Our experimental results demonstrate that the hybrid switch is capable of transferring both predictable traffics and un-predictable traffics successfully. By dynamically selecting the proper switching technique based on the type of communication traffic, the hybrid switch improves communication for most types of traffic.

This dissertation makes five contributions.

1. The first contribution (Chapter 3) is the innovative hybrid switch architecture. Specifically, this architecture combines wormhole switch and predictive circuit switch using virtual channels.

2. The second contribution (Chapter 4) is optimizing virtual channel assignment schemes, by which network bandwidth can be partitioned according to the traffic statistics in order to improve network bandwidth utilization.

3. The third contribution (Chapter 5) is a hardware implemented fast greedy scheduler.

4. The fourth contribution is the near-optimal scheduling algorithm for single crossbar network (Chapter 6) and fat-tree interconnection network (Chapter 7). The proposed scheduling method provides high schedulability ratio with minimum extra hardware cost.

5. The fifth contribution (Chapter 8) is a cycle-accurate network simulator, which provides a uniform simulation platform to evaluate the performance of existing switching techniques, as well as the hybrid switching technique.

1.1 MOTIVATION

1.1.1 Technology impact on parallel computing systems

Communications within a high density are very important. The packaging of parallel computing system consists of a number of racks. Currently, one rack is able to contain hundreds to thousands of processor nodes. As the density of processor package increases; the length of the link connecting a certain number of processors decreases. As shown in Figure 1.2, in IBM BlueGene/L, the design calls for 2 nodes per compute card, 16 compute cards per node board, 16 node boards per 512-node midplane of approximate size 17"x 24"x 34", and two midplanes in a 1024-node rack [10]. Ultra high density blade server is able to fit 336 blade servers into one 42U rack [11].

For short links, it is proposed using copper solutions over optical solutions because of the density of transistors and low cost [12]. In one of the most popular commercial parallel computing networks, Quadrics QsNet II, both optical and Low-Voltage Differential Signaling (LVDS) links are implemented. For links over 13 m in length, a fiber option is advocated; for links under 13 m in length, LVDS links are used [13]. LVDS offers gigabit data rate while consuming significantly less power than competing technologies [14]. LVDS uses a dual serial wire system, running 180 degrees of each other. This enables noise to travel at the same level, which can be filtered more easily and effectively.

In addition, LVDS provides other benefits, among which the most attractive features are very short propagation delay and fast configuration. These features greatly facilitate communications over short links. The propagation latency of an LVDS switch is 1 ns, which is approximately the delay of one foot of cable. Table 1.1 gives two sets of parameters that are typical of digital switches, LVDS switches and optical switches [15, 16, 17, 18].

LVDS channels and LVDS switches are inexpensive. LVDS channels provide a low noise and low amplitude method for gigabit per second data transmission over copper wires [14]. Thus, multiple wires and crossbar switches can be parallelized to achieve higher bandwidth.

With the extremely small latency for LVDS switches, a very fast hardware scheduler, and short links, the overhead for circuit switch is minimized and the benefits are increased.

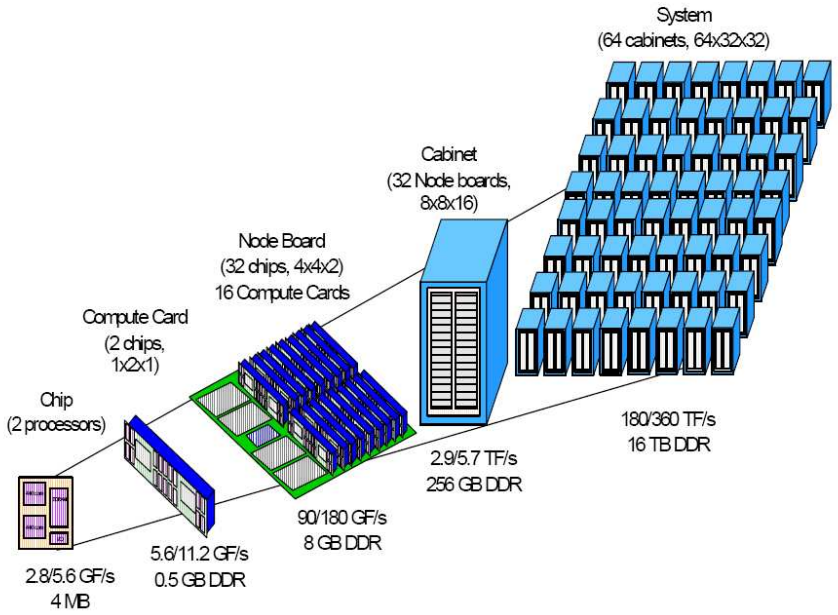


Figure 1.2: Bluegene/L packaging.

Table 1.1: The switch fabric component.

Performance	**Digital**	**Electrical (LVDS)**	**Optical**
Latency	$>$ 10ns	10-50ns	1,000+ns
Propagation Latency	10ns	1ns	1ns
Throughput	100-500Mbps	2.5Gbps	1-10Gbps

If the latency for setting up circuit connections can be eliminated, we can expect even more benefit.

1.1.2 Communication locality and regularity in parallel computing networks

The data traffic in parallel computing systems has its unique characteristics. Messages can be classified as short messages and long messages. In MPICH, the message size varies from 0 bytes to 30,000 bytes [19]. The 80/80 rule states that 80% of the message on the network is 256 bytes or less and 80% of the data on a network is 8,000 bytes or greater [20]. For the NASA Advanced Supercomputing (NAS) parallel benchmark, 80% of the messages are 27,040 bytes or less, and 80% of the data is of a message length of 184,960 bytes or greater. Although the long messages have a small number, the total data inside the long message dominate the communication delay.

The communication on parallel computing network can be divided into two main types, point-to-point communication and collective communication. A point-to-point communication involves one pair of sending and receiving nodes. Collective communication transmits data among all processors in a group specified by an intra-communicator object [21]. It has been found that many parallel applications exhibit communication locality, both in temporal and long term [22, 23, 24]. Collective communication is very regular and its communication pattern is highly predicable. The regularity and locality are helpful for generating efficient network configurations.

1.1.3 Any single switching technique is not sufficient

In order to compare different switching techniques, we have built a uniform simulation platform to evaluate their performance. The switching techniques that are simulated include circuit switching, wormhole routing and predictive circuit switching. Predictive circuit switching is a newly introduced technique, in which the connections are setup before they are requested.

Wormhole routing and circuit switching both have advantages and drawbacks. In the first case, we have simulated a communication system of 32 nodes with random traffic. The

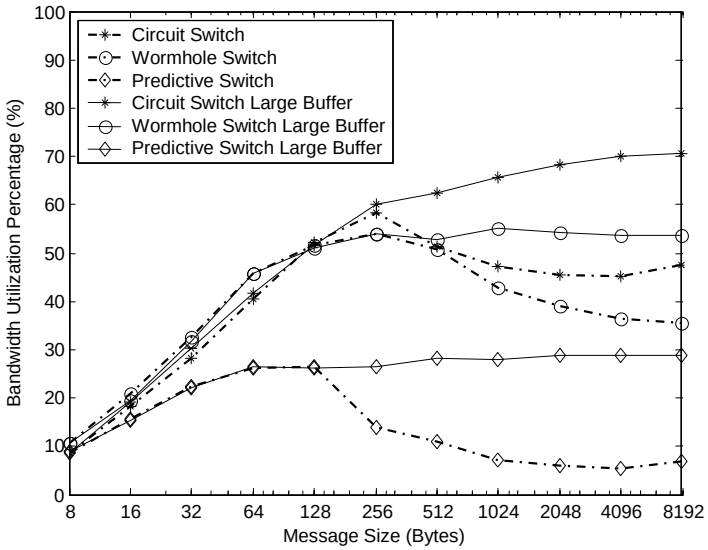
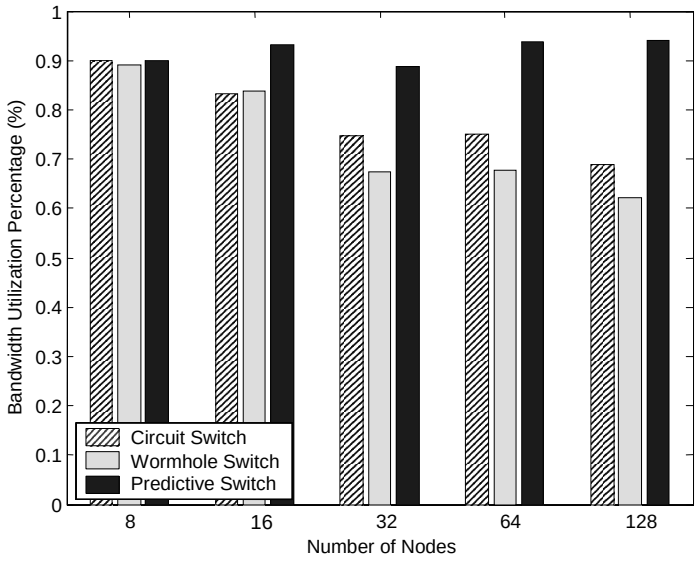


Figure 1.3: Random-to-all communication pattern.

scheduler's working frequency is set to 100MHz. The data generation speed is 1.2Gb/s. The throughput of the network is set to 6.4Gb/s. The buffer inside a NIC is 240KB. The cable latency is set to 100 nanoseconds. These latency approximates 10 foot cables between a NIC and the switch. The performance of the three different networks is shown in Figure 1.3. The dashed line represents the individual system with small buffers, and the solid line represents the system with large buffers. We have found that for small messages, predictive circuit switching and wormhole routing are close in performance. However, wormhole routing is not good for long messages when compared to circuit switching. That is because in wormhole switching, long messages are broken down into packets, and then transmitted as flits; connections are established and removed for each packet. The performance of predictive circuit switching is not satisfactory because the random traffic has very low predictability.

1.1.4 Prospective of predictive circuit switching

Another case shows that predictive circuit switch is capable of making good use of the regularity in parallel programs. Figure 1.4 shows the simulation results for an *All-to-all*



(a) 512 byte message.

Figure 1.4: All-to-all communication pattern.

traffic pattern. When the number of nodes is small, predictive circuit switching, circuit switching and wormhole routing have similar performance. The benefit of predictive circuit switching becomes more pronounced as the system size increases. We suggest building a new switching architecture to combine the advantages of predictive circuit switching and wormhole routing. This enables us to pre-configure the network according to the regularity and locality information extracted from the parallel programs.

This dissertation will focus on a scalable switching architecture to improve network utilization. Our mechanism is to select proper routing algorithms for different types of communications. A new switching technique, virtual channel hybrid switching, is proposed. It takes advantage of both wormhole routing and predictive circuit switching. We propose to combine predictive circuit switching and wormhole routing by virtual channels. With the aid of virtual channels, the ratio of switching techniques can be chosen adaptively.

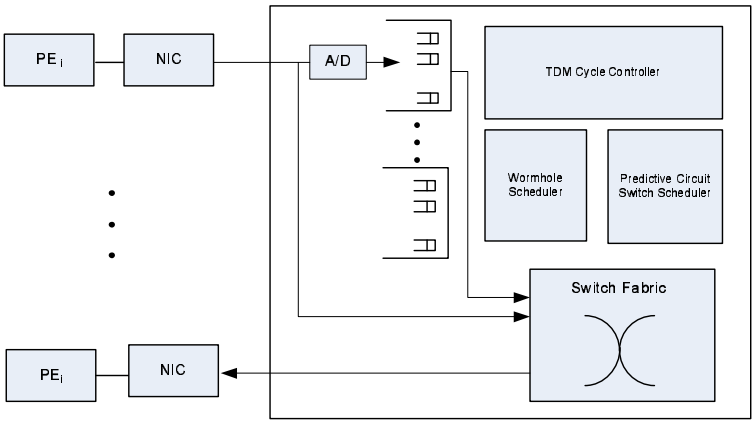


Figure 1.5: Hybrid switching network.

1.2 OUTLINE OF THE DISSERTATION

This dissertation proposes a hybrid switch technique combining wormhole switch and circuit switch using virtual channels. The hybrid switching network architecture is shown in Figure 1.5. The *wormhole scheduler* generates wormhole connections for wormhole traffic and the *predictive circuit switch scheduler* establishes circuit connections for predictable traffic. The circuit connections and wormhole connections are controlled by the *TDM cycle controller* for network configuration. Therefore, two critical issues, which are virtual channel assignment and network scheduling, are investigated. We propose virtual channel assignment schemes to combine wormhole switched traffic and circuit switched traffic. An greedy scheduler is designed to make real-time network configuration. Based on the greedy scheduler, an optimizing scheduler is proposed to make optimal network configuration. We future extend our research to interconnection networks. A Level-Wise scheduling algorithm is designed to configure fat tree networks efficiently. Our experiments show that the hybrid switch performs well under traffic with high predicability as well as under traffic with low predictability. An outline of each chapter is as follows.

Chapter 2 reviews prior research on traffic predictions and current switch technologies. Work that has been done on switch designs combining analog and digital domains is introduced. Related work on extensions of original circuit switching is also presented.

Chapter 3 describes the innovative hybrid-switch architecture. Network components are presented. Two schemes, scheduling predictive request and pre-loading configurations, are used to schedule predicted traffic.

Chapter 4 shows that the performance of hybrid switching is improved by efficiently assigning virtual channels. Without a proper virtual channel assignment scheme the performance of the hybrid switch will decrease under low to moderate predictable traffic. In order to make the hybrid switch work well without very accurate knowledge of traffic patterns, three schemes are proposed for virtual channel assignment.

Chapter 5 presents a fast hardware greedy scheduler which facilitates the hybrid switch architecture. Hardware implementation details are given. A network simulator used to evaluate network performance for this dissertation is introduced briefly. System performance is evaluated for a variety of traffic loads.

Chapter 6 describes an optimizing scheduling algorithm that improves the effective bandwidth utilization in predictive circuit switching. Specialized processors are presented to perform the bipartite maximum matching algorithm for crossbar network scheduling.

Chapter 7 introduces a scheduling algorithm, called Level-Wise scheduling algorithm, to schedule fat tree interconnection networks. Shedulability ratio is defined as the scheduled requests over total communication requests and is used to evaluate performance of scheduling algorithms. The Leve-Wise scheduling algorithm provides a scheduling ratio about 30% higher than general local scheduling algorithms. A hardware implementation architecture for hardware implementation is proposed.

Chapter 8 introduces a framework for design, synthesis and cycle-accurate simulation for parallel computing networks. Modular components are built and intergrated to form networks applying various switching techniques.

This research is concluded in Chapter 9. Future research directions are proposed.

2.0 BACKGROUND AND PRIOR WORK

This chapter reviews prior work that are closely related to the dissertation research. Section 2.1 gives background on compiled communication and traffic prediction. It provides a crucial sumption of the dissertation research. Section 2.2 and Section 2.3 present research and products that utilize a type of hybrid switching. The contributions and issues that may exist in prior work are highlighted at the end of each section.

2.1 COMMUNICATION LOCALITY AND REGULARITY

Communication *regularity* means the pattern of the communications is predictable. If some communication nodes are used frequently within a given period of time, it is said that the communications have *locality*. Some useful research on the prediction of communication patterns both at compile-time and run-time have been done over the past ten years.

2.1.1 Compiled communication

Yuan, Melhem and Gupta [22, 25] analyzed the communication pattern in parallel computing applications using compiled communication. They classified communication into three types: *static communication*, *dynamically analyzable communication* and *dynamic communication*. Static communications are communications whose pattern can be determined at compile time; dynamically analyzable communications are communications whose pattern can be determined at run time without incurring excessive overhead; Dynamic communications are communications whose information can only be determined at runtime.

Static communications and dynamic analyzable communications are a large portion of all communications.

The authors built the E-SUIF compiler that incorporates necessary algorithms into the SUIF [26] compiler tools to support compiled communications. E-SUIF performs communication analysis to get communication patterns on logic processor grids. Then, with the logical communication patterns, physical communication analysis is able to get communication patterns on physical processors. The compiler is also capable of partitioning a program into phases.

2.1.2 Run-time traffic prediction

Afsahi [27] investigated a pattern predictor to predict communication patterns of each separate node. He proposed and compared two types of predictors, cycle predictors and tagging predictors. A cycle predictor is based on the assumption that if a sequence of communication appear once, it may appear later. The communication pattern that happens more frequently has a higher probability to occur in the future. The tagging predictor is performed with the help of a compiler. To insert tags into the original parallel program to indicate a particular section of a code. Each tag is associated with a communication pattern. Once a tag is executed, a certain communication pattern is predicted.

Sakr, Levitan, Chiarulli, Horne and Giles [28] proposed a machine learning model to predict memory access pattern for shared memory multiprocessors. They gave three prediction methods, which are a Markov predictor, a linear predictor and a time delay neural network predictor.

In the Markov predictor, the conditional probability of accessing memory modules is calculated. First order and second order Markov predictors are introduced. For a first order predictor, probability P_{ij} is calculated as the probability of accessing memory i after accessing memory j. For the second order predictor, probability $P_{i(jq)}$ is considered as the probability of accessing memory i if the current processor accesses memory j after accessing memory q. Probabilities are updated after every memory access.

In a linear predictor, memory access vectors are used to present the memory access status. If the i^{th} memory is accessed, the i^{th} bit in a memory access vector is set to one. The linear predictor attempts to predict the next memory access based on a linear combination of all the values of memory access vectors in their history.

A time delay neural predictor predicts the next memory access pattern by taking into account of memory access vectors. The predictor uses tapped delay lines to learn an access pattern. In this way, a particular output pattern can be recognized if it has been seen before, or in response to a specific input pattern. Also, it enables the predictor to generate a complete pattern when a part of the pattern appears.

From these research, we conclude that:

1. Communication in parallel computing system has locality and regularity.
2. Communication pattern can be predicted at compile-time or run-time.

2.2 SWITCH DESIGNS COMBINING ANALOG AND DIGITAL DOMAIN

Over the past years, technologies in ASIC design, serial interfaces, and the emergence of high-capacity and cost-effective analog switches have promoted combining analog and digital technologies into a hybrid switch.

2.2.1 HYPASS: An optoelectronic hybrid packet switching system

Authurs, Goodman, Kobrinski and Vecchi [29] proposed the HYPASS switch architecture. The HYPASS network is formed by two networks: the internal data transport network and the internal control network. The basis of the internal control network is an optical crossbar created from an $N \times N$ passive optical star coupler. This optical switch is configured by the control network. Data are sent as packets and stored at the input port of the optical switch. Packets are transmitted through the optical switch when connections are established.

When a packet arrives at an input port, it is converted to electronic signal by an O/E converter and stored in a buffer at the input port. The packet header is analyzed by the

address decoder. Based on the destination address, wave-length transmitter is tuned to a wavelength that corresponds to the required destination. The internal control network receives the communication request and makes arbitration according to the information gathered from the output buffer. If the request is granted, then a 'send' signal is used to inform the input buffer, and the packet is transferred through the internal data transport network to its destination. After that, a 'data received' acknowledgment is sent to the input port through the internal control network.

2.2.2 HFAST: Hybrid flexibly assignable switch topology

Hybrid Flexibly Assignable Switch Topology (HFAST) [5] provides another solution that combines optical switches and digital switches. Packet switches are grouped as blocks connected through a large-sized optical crossbar switch. Their idea is to configure the circuit switch as a simple topology, such as a 3D torus, to reduce the number of packet switch blocks that need to be traversed.

Unlike HYPASS architecture, the optical switch is placed between nodes and packet switches. Nodes are connected to a passive optical switch crossbar directly. When a source node sends out a packet, the data is transmitted through optical switch. As soon as the data reaches the packet switch blocks, it may be routed to another packet switch block or to another node depending on the connectivity of the optical switch. After one or more hops, the data will reach their destination.

2.2.3 Optical-electrical-optical and optical-optical networks

Livas, Hofmeister and Horne [30] proposed to take an optical-switch approach by combining Optical-electrical-optical (OEO) and optical-optical (OO) switching systems in order to build optical networks over current technology, since a pure OEO system is expensive while OO system only works in wave-length transparency domain. In their proposed architecture, data can be transmitted through either OO system via circuit switching or OEO system via packet switching.

From the above switch designs, we conclude that:

1. Combing digital and analog domain is promising in network design. Electronics components are used for memory and logic control; analog switch fabric is used for data transmission.
2. In the above design examples, packet switching and circuit switching techniques are performed to configure different parts of the network. In the HYPASS and HFAST systems, the packet switch sends and receives packets at the port of the optical crossbar; while circuit switching technique is applied on the data transmission through optical switches. In Livas and his colleagues' design, a packet switch works on OEO systems; while a circuit switch works for the OO system.
3. A potential problem of HFAST system is that a circuit switch configured as a simple topology may not be able to satisfy all communication requests. Some other way has to be found to deal with traffic that does not follow the specific topology.
4. Analog switches only provide a inexpensive and high-throughput crossbar fabric. Without the support of an efficient network configuration mechanism, the crossbar fabric will be under-utilized.

2.3 HYBRID SWITCH EXAMPLES

Due to the overhead caused by establishing physical connections in circuit switching, research has been carried out to improve the circuit switching mechanism.

2.3.1 Configure network for certain communication pattern

The Quadrics interconnect (QsNET) provides special hardware support for multicast [31, 32]. The switch node is able to send a message to any contiguous port in the switch node. The hardware barrier implemented by Quadrics is based on hardware multicast. The root processor sends out a 'test' broadcast packet. The replies of all other processors are combined at the root. After all processors have responsed, a packet is broadcasted indicating the event.

In the Thinking Machines Corporation. CM-5 network [33], regular communication patterns are scheduled. The typical communication patterns pre-scheduled in CM5 are linear exchange, recursive exchange, balanced exchange and pairwise exchange. For example, in pairwise exchange, if the system size is N, each processor takes $N-1$ steps to receive messages from all other processors. During step i, processor n exchange a message with processor $(n+i)modN$.

2.3.2 Configuration for large messages

Related research has been done on applying different switching algorithms for messages of different sizes. Coll and his colleagues [34] proposed a hybrid algorithm for networks in high performance clusters. Two algorithms, which are *local dynamic allocation algorithms* and *dynamic allocation algorithms*, are proposed to handle traffic. In the local dynamic allocation algorithms, the NIC only collects local information and starts sending data depending on the local transaction requirement. In the dynamic allocation algorithms the local and remote information are checked. The goal is to guarantee that both the sender and the receiver are free before sending data. The local dynamic allocation algorithms are used to switch short messages and the dynamic allocation algorithms are used to transfer long messages. Specifically, if local NIC is available and requests a connection, a *Request To Send*(RTS) is sent to the destination NIC. Upon receiving the RTS signal, the NIC of the destination replies a *Clear To Send* (CTS) or *Negative Acknowledgment* (NACK) to indicate the available or unavailable status. This is very similar to a circuit switching scheme. Since the rail reservation using dynamic allocation introduces an overhead for every message, the authors recommend applying the dynamic allocation on only large messages.

Chen and Liu [35] presented a hybrid network architecture for database and multimedia systems of hypercube topology. They integrated self-routed wormhole routing and circuit switching based on virtual channels. They divided the channels into a short message channel and a long message channel to serve different types of messages. L is set as a threshold of message length. Messages shorter than L are named as short message; messages longer than L are long messages. The short messages are routed deterministically using wormhole

routing. The routing path for long message is determined globally. To route a long message, a *probing* signal is initiated by the source node first. The probing signal propagates through the network in short message channels following depth-first rule to establish connections. If the destination node receives the probing signal, a success *acknowledging* message is sent back to the source node via short message channels. In this way, a long message path is established.

2.3.3 Wave switching

Duato, Jose, Lopez and Yalamanchili [8, 9] proposed wave switching that combines wormhole switching and circuit switching. When two nodes are communicating frequently, the physical circuit connection will be kept for a certain time. If the router contains switches, S_0, ..., S_K. The physical channels on switch S_1, ..., S_K are used for data transmission through circuit switching. The physical channels in S_0 are split into $k + w$ virtual channels. K channels are used to transfer control flit, which are used to set up and tear down connections on switches S_1, ..., S_K. The remaining w virtual channels are used for wormhole switching.

This design is promising, except that the partition of physical channel for circuit switch and wormhole switch is fixed. If there are not enough circuit switched traffic, switches S_1, ..., S_K will be left unused.

From these papers, we conclude that:

1. Researchers have shown that packet/wormhole switching and circuit switching can be complimentary.
2. Network performance improves when configuring the network as a certain network topology or for a regular communication pattern.
3. However, not all predictable traffic follows a certain topology. The scheduling method to efficiently schedule traffic is crucial.
4. Better schemes need to be investigated to handle both predictable traffic and un-predictable traffic.

3.0 HYBRID SWITCH ARCHITECTURE

A description of the hybrid switch architecture is given in this chapter. Section 3.1 introduces current switching techniques, including circuit switching, packet switching and wormhole switching. Message latency in switched interconnection networks is analyzed. Section 3.2 describes the hybrid switch components. In Section 3.3, the communication modes by which the hybrid switch handles both predictable traffic and un-predictable traffic are discussed in detail. Section 3.4 provides detailed architecture of network components.

3.1 CURRENT SWITCHING TECHNIQUES

As we have mentioned in Chapter 1, the major switching techniques are circuit switching, packet switching and wormhole routing. Circuit switching establishes an entire source-to-destination route before any data are sent [36]. Establishing this route incurs a high latency cost. Each connection occupies one physical link and once the circuit is established, it may block other circuits from forming. The benefit is in the simple switching elements, as they do not need to contain any data buffering and only need enough logic to determine their current configuration. As soon as the connection is established, the message can be transferred with low latency. For example, a 4×4 LVDS switch has a propagation delay of 1 ns.

Figure 3.1 shows a circuit switched network. T_w represents delay of wires between a NIC and a switch. The scheduler latency is denoted as T_sch. T_setup represents the delay caused by switch configuration. If a message is sent from PE_i to PE_j, the message is buffered in a network interface card (NIC) after it has been generated by PE_i. The NIC sends a request to a network scheduler. After T_sch, a grant is sent back to the NIC if

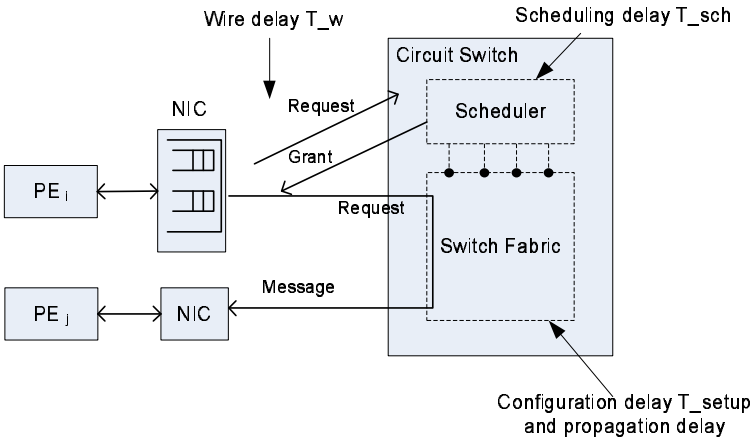


Figure 3.1: Message delay in circuit switched network.

a connection is successfully scheduled. The NIC injects the message to the network after receiving the grant.

The latency, T_ckt, of a message transmitted through a circuit switched network is represented as below, where $T\_message$ = (size of a message)/(switch's throughput).

$$T\_ckt = T\_sch + T\_setup + 4T\_w + T\_message \tag{3.1}$$

Packet switching sends data payloads of limited size through the network by adding routing information to the front of the payload, thereby creating a data packet. Each data packet is independent of others [37, 38, 39]. When a large amount of data needs to be communicated, multiple packets are created and sent through the network. The packet switch is more flexible than circuit switching because the possible connections are not limited by the number of the physical links. However, the packet data transmission is based on the assumption that the entire packet has to be received before forwarding it out. This introduces buffering latency.

Figure 3.2 depicts the procedure of a packet transmission in a packet switched network. After PE_i generates a message, the NIC connected with PE_i breaks down the message into K packets, where K = (size of a message) / (size of a packet). Each packet is stored in a buffer inside a switch before being forwarded to its destination. After detecting a complete

paacket, the switch port sends out a request to the switch scheduler asking for a connection. After T_sch, a grant is received by the buffer if the connection can be established. The packet is forwarded to its destination. The connection is torn down after the packet is transferred.

The latency T_pkt of a message transmitted through a packet switched network is shown as below,where T_packet = (size of a packet)/(switch's throughput),
k = (size of a message)/(size of a packet), and $T\_buffering$ is the time used to store entire packet.

$$T\_pkt = 2T\_w + k(T\_sch + T\_setup) + T\_buffering + T\_packet \tag{3.2}$$

Wormhole switching improves on packet switching by establishing a path through the network as it is routed. Referring to Quadrics network [40], a well known wormhole switched network, we define the wormhole routing as to be used in this dissertation. In wormhole routing network, a packet is broken into several flits, which is the unit for storing and forwarding. The head of the worm establishes the route though the network and all subsequent flits take the same path. The intermediate switching node uses less buffer space and does not wait for the whole packet. Therefore, the big buffering problem in packet switching is removed. The latency is also illustrated in Figure 3.2.

The procedure of message transmission using wormhole switches is similar to that using packet switching. The major difference is that a buffer generates a request upon detecting a flit instead of a complete packet.

The latency T_worm of a message transmitted through a wormhole switched network is represented as the following:

$$T\_worm = 2T\_w + k(T\_sch + T\_setup) + T\_packet \tag{3.3}$$

Time division multiplexing (TDM) switching [41, 42, 43] is an extension of circuit switching in which the switch alternates between K configurations, where each configuration establishes circuits between the inputs and the outputs of the switch. Hence, a particular connection, is established every K time slots and thus receives $1/K$ of the bandwidth, where K is the multiplexing degree. In other words, scheduling a connection on a TDM switch means scheduling it repeatedly, on any one of the K multiplexed slots. If the circuit connection is established and torn down for each message, the message latency is the same as

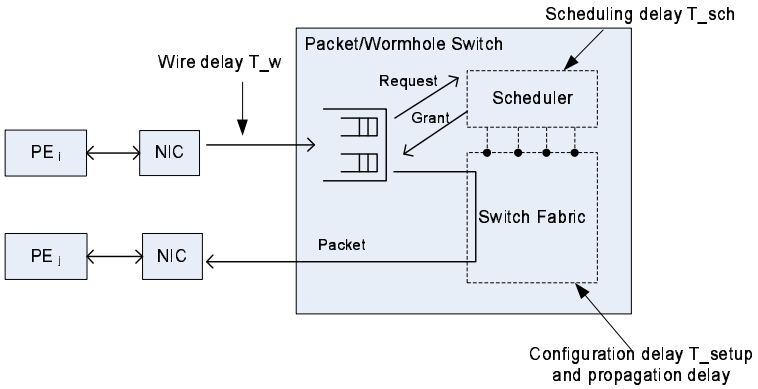


Figure 3.2: Message delay in packet/wormhole switched network.

that of circuit switching. We incorporate the TDM method into our hybrid switching in a different way in order to amortize the scheduling overhead, which is discussed in Chapter 5.

The design of a network scheduler is crucial to reduce message latency. It is observed that the wire delay T_w, the switch configuration time T_setup and the scheduling delay T_sch impacts the message latency in all switched networks. T_w is determined by the scale of networks. T_setup, $T\_message$ and T_packet are decided by current switch fabric technologies. However, it is practical to reduce scheduling latency with innovative hardware design. According to Equation 3.1, 3.2 and 3.3, reducing the scheduling latency T_sch benefits circuit switching, as well as packet switching and wormhole switching.

3.2 HYBRID SWITCH COMMUNICATION SYSTEM

Based on the analysis of message latency in communication networks using various switching techniques, we propose a predictive circuit switching technique. Predictive circuit switching amortizes scheduling latency by pre-establishing and re-using connections. Our hybrid switching combines predictive circuit switching and wormhole switching in order to take advantage of both switching techniques.

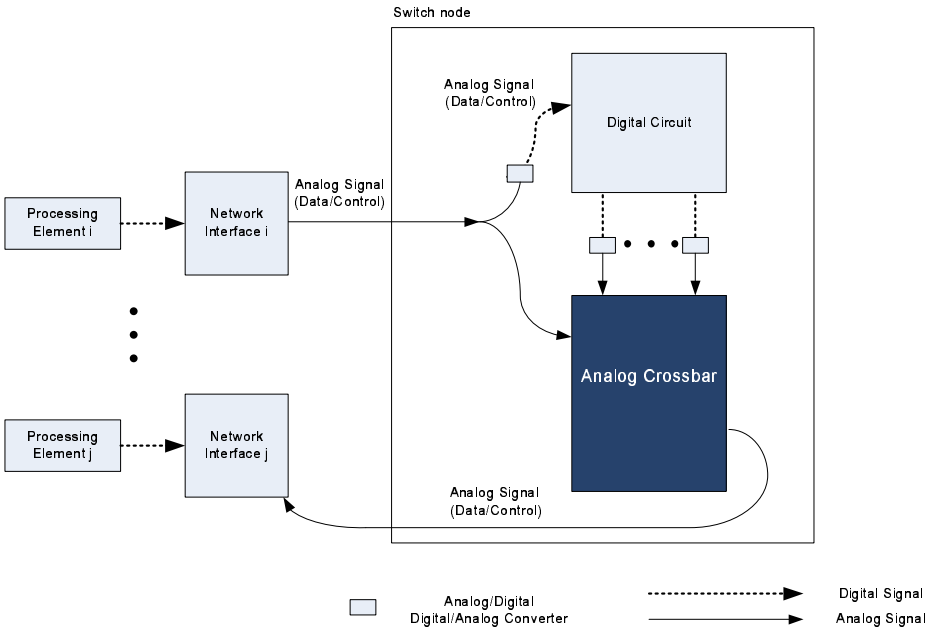


Figure 3.3: The hybrid switching system.

The overall hybrid switch architecture and switching mechanism are described in this section.

3.2.1 Hybrid switch system

Our proposed a hybrid switch systems combines wormhole switch and predictive circuit switch. Predictive circuit switching is motivated by the observation that a portion of communications within parallel computing has regularity and is highly predictable and thus, can be deterministically scheduled. This is a variation on circuit switching in which the switch is scheduled based on the prediction of the communication requirements of the application. The detailed architecture of the predictive circuit switch is shown in Chapter 5. In this section, the overall architecture of the hybrid switch is presented. The proposed hybrid switch consists of four main components, as shown in Figure 3.3.

Figure 3.4: Data format in the hybrid switch.

1. Processing elements (PE)

   The processing elements are computation nodes in the parallel computing system. Each processing elements is coordinated with other processing elements through network interface cards to provide powerful computation capability.

2. Network interface (NIC)

   Each processing element is connected with a network interface card. Data is injected into network interface card and stored inside it for network transmission. Control data that represent traffic patterns, as well as communication requests and grants, are also input and output signals of the network interface card. Communication data and control data share the same high-speed link connected between the network interface cards and the switch node.

3. Switch node (SW)

   The switch node is responsible for scheduling and enabling network communications. The switch node is an analog and digital mixed signal design. A digital circuit is used for buffering and scheduling wormhole switched traffic. The analog switch is configured by the digital circuit and its bandwidth is shared by wormhole switched traffic and the predictive circuit switched traffic.

3.3 COMMUNICATION MODES

The hybrid network architecture supports both predictable traffic and un-predictable traffic and operates in three modes, un-predictable traffic mode, predictable traffic mode and preloaded mode.

In each communication mode, data transmission is initiated by a processing element and requests a sequence of actions executed by network components.

Figure 3.4 shows the data format used in the hybrid switching system. A 'Tag' is used to indicate the traffic type. If the 'Tag' in a message is one, it indicate the message is a predictive circuit switched message. Otherwise, it is a wormhole switched message. No matter what switch method will be used to transfer the message, the message will be sent to both the digital circuit and the analog switch. For a predictive circuit switched message, its 'Tag' bit is '0'. A connection has been established for it in the analog switch, so the message is transferred through the network. Meanwhile, in the digital circuit, the 'Tag' bit is detected. All messages whose 'Tag' equals one are discarded. For a wormhole switched message, its 'Tag' bit is '1'. Because no connection is established for it, it will be dropped by the analog switch even it is injected at the input port. The digital circuit detects that its 'Tag' equals '1', therefore stores the packet for wormhole switching in the future.

The detailed data transmission process is described in the following.

System variables are defined as below.

K: Number of time slots for predictive circuit switching

N: System size

PE_i: The i^{th} processing element

NIC_i: The i^{th} network interface card

3.3.1 Unpredictable traffic mode

The *un-predictive traffic mode* is similar to general wormhole switched communication. Data are stored and forwarded in the switch node. Connections are established and torn down for each packet. A simplified system architecture is used to illustrate the un-predictable traffic communication, as shown in Figure 3.5.

1. PE_i generates a message to PE_j.
2. NIC_i receives the message and checks the destination of the message in the predictive circuit switch table. If the required destination is not set for predictive circuit connections, then the tag bit is set to '1' to indicate the message as wormhole switched traffic.

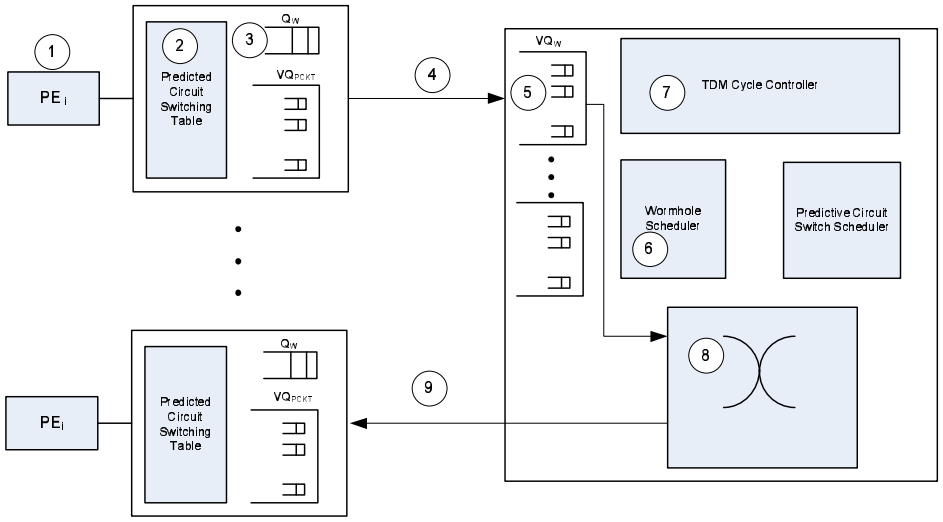


Figure 3.5: The un-predictable traffic communication mode.

3. The message is stored into wormhole buffer Q_W.
4. If the wormhole buffer VQ_W in the switch node is not full, NIC_i forwards the message to switch node and stores it in VQ_W.
5. VQ_W generates requests to the wormhole scheduler. The wormhole scheduler arbitrates in round-robin fashion.
6. If the request from PE_i to PE_j is granted, the connection is added wormhole connections W and stored in the wormhole configuration registers.
7. The TDM cycle control generate the time slot ID, if current ID number equals K, the wormhole connection is selected as the output, and W is sent to VQ_W.
8. Upon receiving W, the message is output from VQ_W and sent out through the analog switch
9. The message arrives NIC_j.

3.3.2 Predictable traffic mode

Data communication through predictive circuit switching is called the *predictable traffic mode*. Circuits are established before predictable traffics come. The pre-established circuit

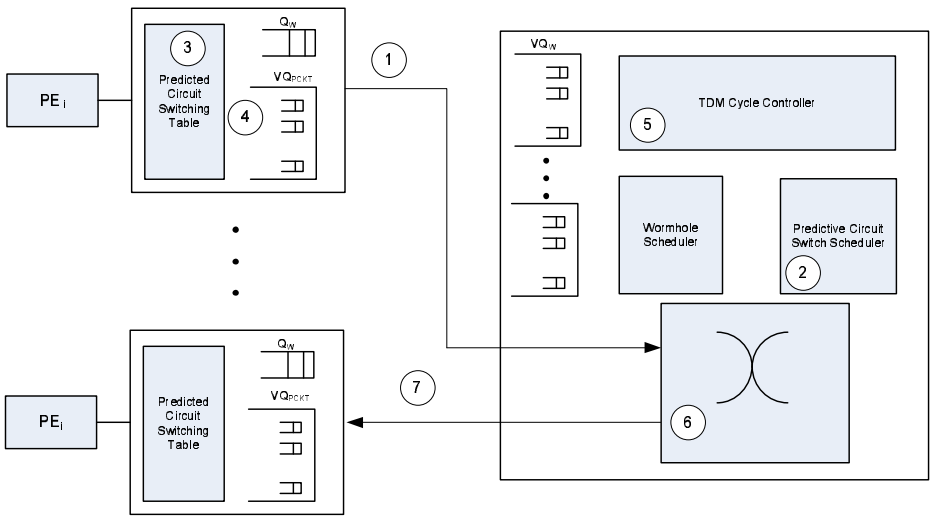


Figure 3.6: The predictable traffic communication mode.

connections are time multiplexed. The circuits is connected and disconnected according to predicted communication requests rather than dynamic communication requests. The predictable traffic communication is divided into seven steps, as shown in Figure 3.6.

1. NIC_i sends out predicted requests to the switch node requiring a connection from PE_i to PE_j.
2. The predictive circuit switch scheduler receives the request and makes scheduling. Let us assume the connection is scheduled at time slot p.
3. The updated predictive circuit switching connections B^* are sent back to NIC_i to update its predictive circuit switching table. B^* is defined as the total connections established for the predictive circuit switching.

4. NIC_i receives a message from PE_i to PE_j. NIC_i detects the destination address in the message that has existed in the predictive circuit switch table. Therefore, the tag bit is set to '0' to indicate the message as predictive circuit switched traffic.
5. The message is stored in VQ_{PCKT}.
6. The TDM cycle controller generates the time slot ID, if the current ID number equals p, the circuit connections $B^{(p)}$ are be selected, and grant signals G are sent to NIC_i.
7. Upon receiving G, the message is output from VQ_{PCKT} and sent out through the analog switch.
8. The message arrives NIC_j.

3.3.3 Preloaded mode

The predictive circuit switch schedule can be preloaded with a set of configurations. The set of configurations are activated one-by-one to control the network. The connections are established before real traffic comes and is not effected by either predicted communication requests or demanding communication requests.

1. NICs send out network configurations for time slot $0, ..., K-1$.
2. Switch node stores the configuration as $B^{(0)}$, ..., $B^{(K-1)}$ in registers.
3. The total grants B^* are sent to NICs to update their predictive circuit switching tables.
4. NIC_i receives a message from PE_i to PE_j. NIC_i detects the destination address in the message has existed in the predictive circuit switch table. Therefore, the tag bit is set to '0' to indicate the message as circuit switched traffic.
5. The message is stored in VQ_{PCKT}.
6. TDM cycle control generate time slot ID, and the circuit connections according to the time slot ID are be selected, and grant signals G are sent to NIC_i.
7. Upon receiving a G containing the connection from PE_i to PE_j, the message is output from VQ_{PCKT} and sent out through the analog switch.
8. The message arrives NIC_j.

The three communication modes work individually or cooperatively to adapt the real communication requests. Assuming that a network switch system contains K time slots,

we can pre-load configurations for P slots, set N slots for scheduling predictive traffic, and set one slot for scheduling un-predictable traffic, as long as $P + N < K$. The proposed hybrid switch architecture is capable of supporting predictive circuit switching and wormhole switching.

3.4 NETWORK COMPONENTS

The NIC and the switch node are important components, thus are described in detail. In order to simplify description of components, the control signal and data signal are shown as separate wires but in reality one physical wire sends both types of data.

3.4.1 Network Interface

We define the following signals for clarity.

D: Message transferred though the network, including D_W and D_{PCKT}

D_w: Messages transferred through the network via wormhole switch

D_{PCKT}: Messages transferred through the network via predictive circuit switch

U: Connections established for wormhole switching

B^*: Total connections established for predictive circuit switching

$B^{(i)}$: Connections established for predictive circuit switching in time slot i

R_c: Communication requests for predictive circuit switching

R_w: Communication requests for wormhole switching

P: Traffic prediction

F: Switch fabric configurations

G_i: Grant signals to the i^{th} NIC

In order to support both wormhole and circuit switched traffic, each NIC contains two buffers as shown in Figure 3.7, one buffer Q_W for wormhole switch and a virtual queue for predictive circuit switch, denoted as VQ_{PCKT}. The virtual queue is a physical buffer, but is logically partitioned into N queues corresponding to N destinations so that head-of-

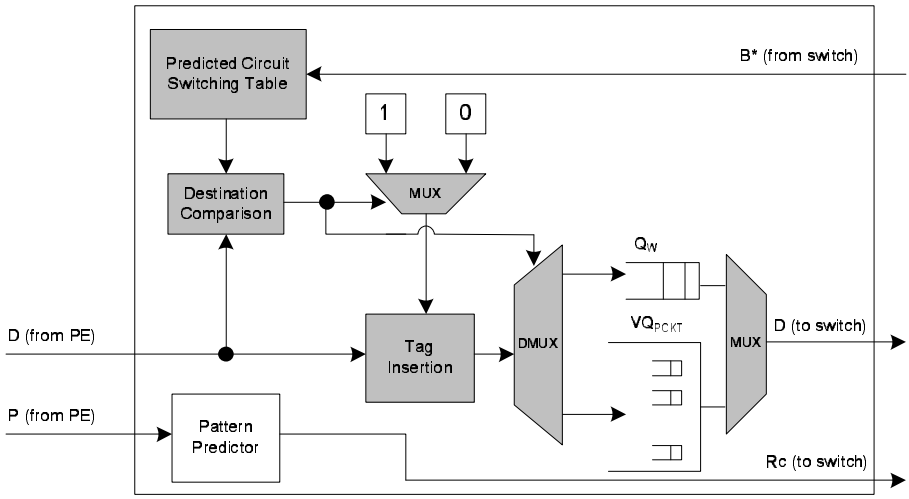


Figure 3.7: The network interface card.

line blocking can be avoided. Since there are no buffers for predictive circuit switch in the switch node, the virtual queue buffer for predictive circuit switching is implemented inside the network interface card. For wormhole switching, most data is buffered in the switch node. Therefore the virtual queue buffers are implemented in the switch node for buffering wormhole traffic following the same idea.

The *Predicted Circuit Switching Table* inside the network interface decides which buffer to use for temporal data storage. The value of the Predicted Circuit Switching Table is an N-bit value. Each entry is an N-bit vector, representing the connections scheduled using predictive circuit switch. For example, if the 3^{th} entry equals '00000101' in a 8×8 system, the vector shows the links from PE_3 to PE_2 and PE_0 have been established in predictive circuit switch. In this way, all data from PE_3 to PE_2 to PE_0 are stored in virtual queues for predictive circuit switching and the rest of the data from PE_3 are stored in the buffer for wormhole switching.

In this dissertation, there are two ways to set the values in the Predictive Circuit Switching Table. The first one is by detecting the grants from the predictive circuit scheduler. By receiving traffic prediction from compiler or run-time predictor, the predicted requests will

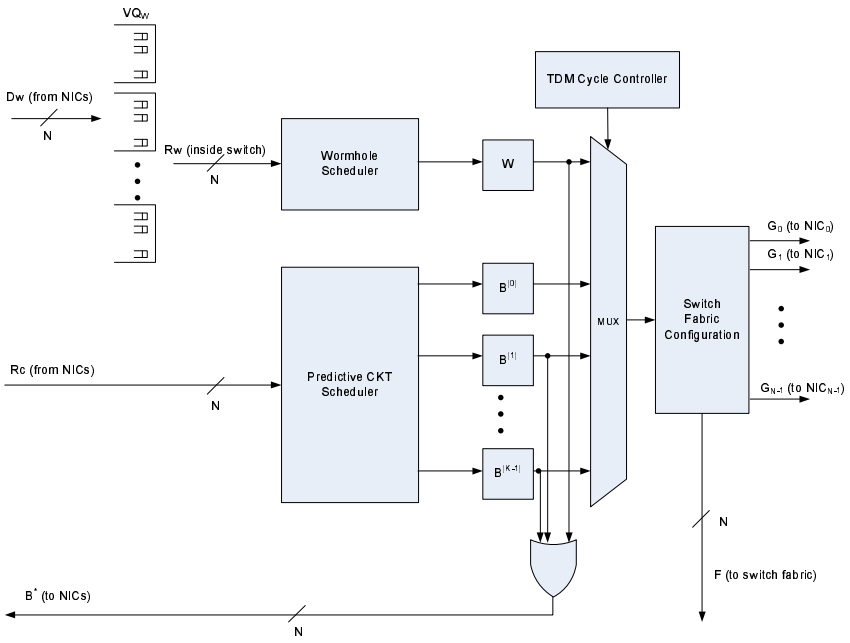


Figure 3.8: Internal architecture of the digital circuit inside the switch node.

be sent to the predictive circuit scheduler. After scheduling, the predictive circuit scheduler sends back granted connections. The values in the *Predictive Circuit Switching Table* are updated according to the granted connections. The second method is named as pre-loading in contrast to on-line scheduling. Configurations are preloaded into the switch node. Meanwhile, the network interface gets the same configuration information. Without waiting for scheduling, the values in the predicted circuit switching table are set instantly. In the former method, not all predicted requests are get granted, therefore the network interface has to wait for scheduling results to set up the values in the *Predictive Circuit Switching Table*. Grants $B^{(0)}$, ..., $B^{(P-1)}$ are combined together using a 'or' logic, where P is the total number of time slots being pre-loaded. The output B^* is returned to NIC to updated value in the predictive circuit switch table.

3.4.2 Switch node

The switch node contains analog switch fabric and a digital circuit. The internal architecture of the digital circuit is illustrated in Figure 3.8. Two schedulers, which are wormhole scheduler and predictive circuit switch scheduler, schedule wormhole traffic and predictive circuit switched traffic respectively. The wormhole scheduler receives communication requests R_w from buffers inside the switch node, denoted as VQ_W. The wormhole scheduler in the hybrid network performs standard wormhole switching. The scheduled connections U are stored in registers. Predictive circuit switch scheduler receives predicted communication requests R_c from the network interfaces. It attempts to schedule communication requests in K time slots, assuming K is the maximum number of time slots that can be scheduled. The predictive circuit switch scheduler starts to perform scheduling from the first time slot, namely time slot 0. If failed, the predictive circuit switch scheduler will try the next time slot until (1) either the request is successfully scheduled (2) or the scheduler reaches the last time slot. The successfully scheduled connections for time slot i, $B^{(i)}$, are stored in registers. Registers, which keep the connections of B^0, ..., B^{K-1} and U, are connected to a multiplexer. A TDM cycle controller generates current time slot ID and selects a set of connections corresponding to the current time slot. The output of the multiplexer are grant signals to network interfaces.

Besides storing results getting from the predictive circuit switch scheduler, the registers are able to set as pre-defined values coming from NICs. In this way, configurations are pre-loaded instead of being scheduled. This feature is very useful when communication follows very clear network topology, such as mesh, torus and trees. The network can be configured just as the network topology by pre-loading different network mappings into different time slots to satisfy predictable communication requirements.

4.0 OPTIMIZING VIRTUAL CHANNEL ASSIGNMENT

Chapter 5 shows the advantage of the hybrid switch, but focuses on traffic that is very predictable. In this chapter, we introduce virtual channel assignment schemes to loosen the traffic prediction constraint. In Section 4.1, background information about virtual channels is introduced and terms are defined for the rest of the chapter. Section 4.2 describes the core of virtual channel assignment. Section 4.3 describes Skip Empty Slots (SES) scheme that enables the switch to improve network bandwidth utilization. SES is set as baseline for the performance analysis. Section 4.4 describes the Slot Length Adjustment (SLA) schemes to allocate slots according to the the traffic bandwidth requirement. The third scheme described in Section 4.5, preempt virtual channels (PREEMPT) enables good network performance for traffics with vague phase boundaries. Performance evaluation is given in Section 4.6. Conclusions are draw in Section 4.7.

4.1 INTRODUCTION

In the hybrid switching system, virtual channels are used to let wormhole switched traffic and predictive circuit switched traffic share one physical switch fabric. In our hybrid switching system, virtual channels are implemented in a Time-Division Multiplexing (TDM) manner.

Data are transferred through switch fabric in a certain period of time, named a time slot. The length of a time slot is called the TDM *slot length*. A group of time slots form a *TDM cycle*, in witch the switch fabric is configured in different ways in different slot. During data transmission, TDM cycles are repeated. Figure 4.1 shows an example of a TDM cycle. The TDM cycle contains K time slots, time slot 0, ..., $K-1$.

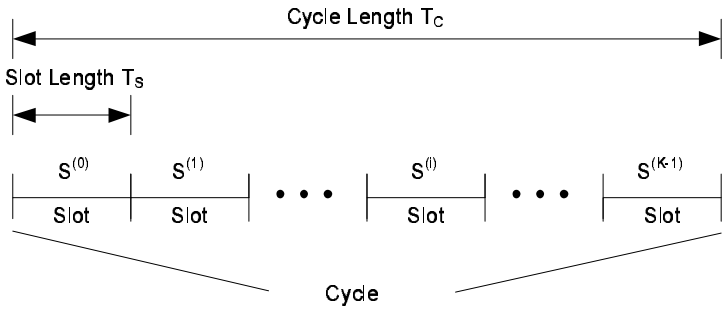


Figure 4.1: Definition of TDM cycle, TDM slot and slot length.

4.2 COMBINE WORMHOLE SWITCHED TRAFFIC WITH PREDICTIVE CIRCUIT SWITCHED TRAFFIC

Wormhole switched traffic and predictive circuit switched traffic are combined within a TDM cycle. Our hybrid switching system contains $K+1$ time slots, $S_P^{(0)}$, ..., S_P^K, allocated for predictive circuit switching and one time slot S_W allocated for wormhole switching.

4.2.1 Virtual channels of the hybrid switch

Network configurations for predictive circuit switching are enabled at time slot $S_P^{(0)}$, $S_P^{(1)}$..., and $S_P^{(K-1)}$, as shown in Figure 4.2. At each time slot, one set of network configurations configures the analog switch for data transmission. Network configurations $B^{(i)}$ for predictive circuit switching are enabled at time slot $S_P^{(i)}$, where $i < K$. Switches are configured for predictive circuit switching at the start point of each slot and the configurations are kept unchanged within a slot. Network configurations for wormhole switching are enabled at time slot S_W. During S_W, the analog switch is controlled by the wormhole switch. Different from the network configurations in $S_P^{(0)}$, ..., $S_P^{(K-1)}$, the network configurations are based on a packet, instead of a time slot.

Figure 4.3 depicts the procedure of data transmission through virtual channels. In Figure 4.3, the data from PE_i to PE_j is represented as '$i \rightarrow j$'. Assume a processing element PE_0

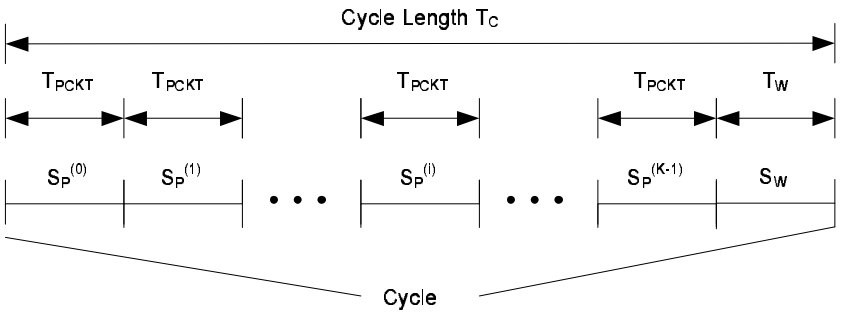


Figure 4.2: Slots in hybrid switching system.

generate traffics to PE_1, PE_3, PE_4, PE_{12}, PE_6, PE_7, PE_8, and PE_9. The data to PE_1, PE_3, PE_4, and PE_{12} is predictable. Assume that the connections for predictable traffics have been established as predictive circuit connections. Assume the connection from PE_0 to PE_1 is allocated at the first time slot $S_P^{(0)}$ and stored in the configuration set $B^{(0)}$; the connection from PE_0 to PE_3 is allocated at the first time slot $S_P^{(1)}$ and stored in the configuration set $B^{(1)}$; the connection from PE_0 to PE_4 is allocated at the second time slot $S_P^{(2)}$ and stored in the configuration set $B^{(2)}$; the connection from PE_0 to PE_{12} is allocated at the third time slot $S_P^{(3)}$ and stored in the configuration set $B^{(3)}$. The wormhole switch performs scheduling by a round robin policy, hence, it establishes connection for the packet to PE_6 and tears down the connection after one packet is transferred. In our example, the connection for the packet to PE_7 is to be established and torn down after the packet is sent. Similarly, the connect and disconnect loop applies to the rest of wormhole switched packets.

During the first cycle, the sequence of slots is $S_P^{(0)}$, $S_P^{(1)}$, $S_P^{(2)}$, $S_P^{(3)}$ and S_W. The configurations of $B^{(0)}$, ..., $B^{(3)}$, and W are enabled by the sequence of slot numbers. As configurations $B^{(0)}$ are enabled, predictive circuit switched data from PE_0 to PE_1 is transferred. The predictive circuit switched data to PE_3, PE_4 and PE_{12} are transferred through the network in the same way in slot 1, 2 and 3. The time slot following $S_P^{(3)}$ is S_W. During S_W, the configurations stored as W are enabled. Configurations W are variational based on the packets buffered. The connections may change within the same S_W slot (but not within S_P). Figure

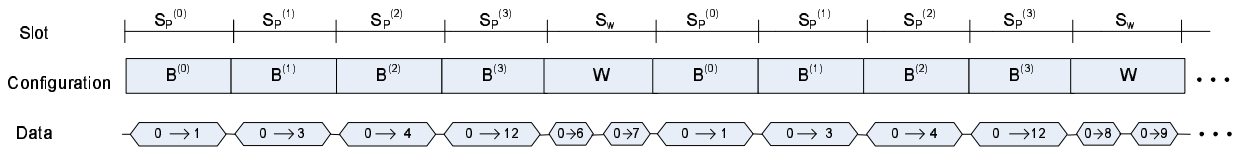


Figure 4.3: Traffics combination.

4.3 shows that two packets are transferred in S_W, one packet is sent to PE_6 and the other packet is sent to PE_7.

During the second cycle, the predictive circuit switched data are transferred through the network following the exact manner as in the first cycle. The wormhole switched traffics are transferred by continuing the round-robin configurations. As it can be seen in Figure 4.3, two packets are sent to PE_8 and PE_9. The destinations of the wormhole switched packet are different.

The network configurations for S_W are decided by the wormhole switch scheduler, while the network configurations in $S_P^{(0)}$, ..., $S_P^{(K-1)}$ are decided by predictive circuit switch scheduler. The way of assigning the virtual channels impacts the network performance. Section 4.3, Section 4.4 and Section 4.5 introduce three methods of virtual channel assignment for the hybrid switch.

4.2.2 Configure virtual channels for predictive circuit switch

Virtual channels assignment focuses on allocating slots for two different switching approaches. Before introducing the virtual channel assignment, the methods that are used to configure virtual channels for predictive circuit switch are reviewed.

1. Assigning virtual channels based on predicted requests

   The most straightforward method to configure time slots for predictive circuit switching is to use the predictive circuit switch scheduler. Predicted request are sent to the scheduler. The predictive circuit switch scheduler starts to allocate network resources from the first time slot. The scheduler allocates network resources from time slot 0 to time slot $K-1$

in order to grant those requests. Each established circuit connections occupies one slot in a TDM cycle. If the schedule is full, unscheduled requests are dropped. Each NIC receives a grant vector that describes its portion of the schedule.

2. Pre-load virtual channels

   The second method to configure time slots for predictive circuit switching is pre-loading. In two cases, pre-loading is highly recommended. The first case is when the communication strictly follows a virtual topology. For example, if a communication strictly follows a 2-D mesh network topology, then it is possible to use four time slots to configure the switch to satisfy all communications. The second case is for collective communications. For example, if the communication is a 'scatter', one processing element sends messages to all other processors. We could configure the switch temporally to fit the communication pattern without using the predictive circuit switch scheduler. The benefit of pre-loading to get optimal configurations for predictable communications. That involves a set of configurations from a group of processors.

The basic TDM cycle controller is implemented by a counter, as shown in Figure 4.4. The output $SLOT$ is $0, 1, ..., K$. The signal $TRIG$ is the output of a clock divider. The output clock frequency of the clock divider decides the length of each slot. There are many ways to design a clock divider. Figure 4.4 gives one implementation. A preset value D_L is loader to a LATCH. A COUNTER generates output D_C to compare with D_L. When the two values are equal, signal Eq triggers FLIP-FLOP to toggle the output.

4.3 SES: SKIP EMPTY SLOTS

Not all $K + 1$ slots need to be fully scheduled and some may be entirely unused. Time slots may be un-used in either configuration method. When the network configurations for predictive circuit switch occupy j time slots, if $j < K + 1$, then $K + 1 - j$ time slots do not contain any network configurations. The time slots assigned for predictive circuit switching or wormhole switch may not contain any network configuration. We define this kind of time

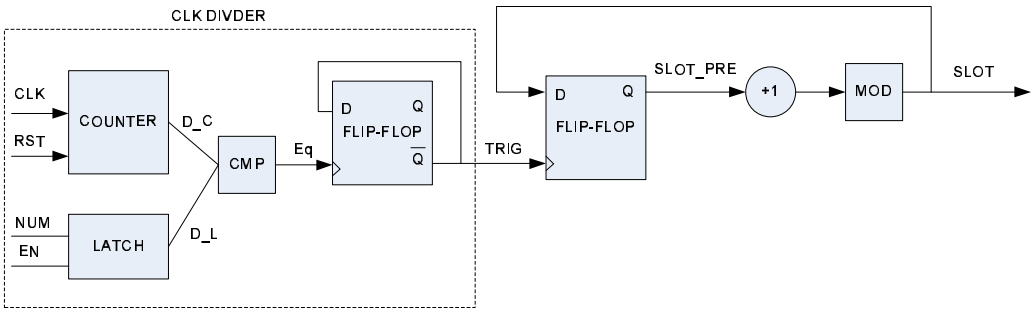
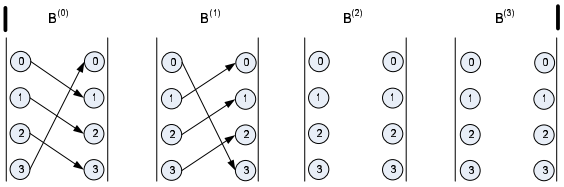
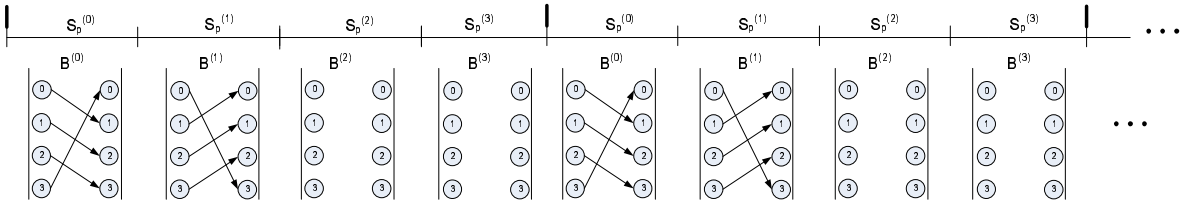


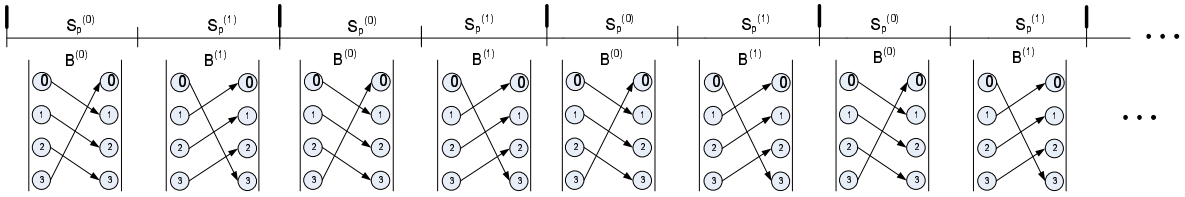
Figure 4.4: TDM cycle controller implemented as a counter.



(a) Network configurations scheduled for predictive circuit switch.



(b) Network configurations without SES.



(c) Network configurations with SES.

Figure 4.5: Network bandwidth utilization gets improved with SES.

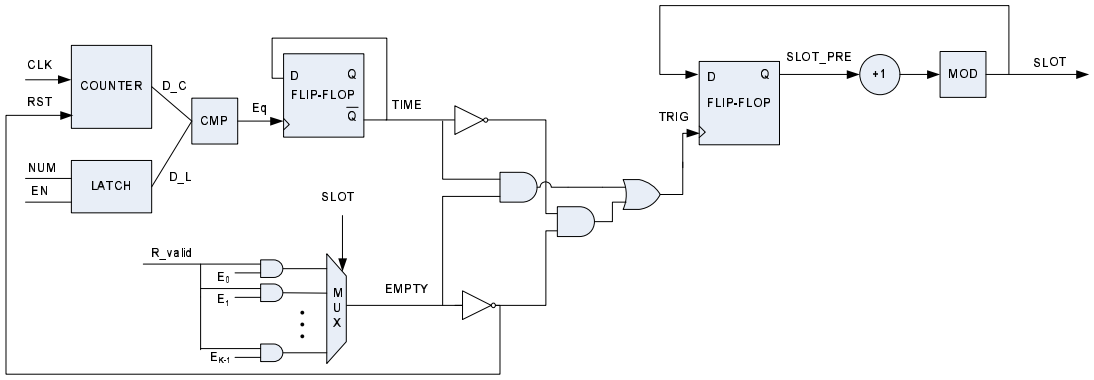


Figure 4.6: TDM cycle controller with SES capability.

slot to be an *empty slot*. During empty time slots, no data are transferred through the network.

Empty slots introduce inefficient network utilization. Figure 4.5 gives a example of configurations scheduled for predictive circuit switch. In order to simply the explanation, we ignore S_W in the cycle. Configurations of set $B^{(0)}$ and $B^{(1)}$ are fully loaded. However, there is no configuration in $B^{(2)}$ and $B^{(3)}$, as shown in Figure 4.5(a). If we treat all slots equally, the TDM cycle controller generates time slot in the normal sequence, which is $S_P^{(0)}$, $S_P^{(1)}$..., and $S_P^{(3)}$. Obviously, no data are transferred through the network at $S_P^{(2)}$ and $S_P^{(3)}$ because the network does not contain configurations during the two slots. In this example, 50% of the network bandwidth is under-utilized.

Moreover, the S_W time slot assigned for wormhole switching is not always used. (1) When the communication is highly predictable, such as collective communications, most of the network traffic goes through predictive circuit switch channels; (2) When the communication follows a certain topology strictly, such as a 2-D mesh, all traffics go through predictive circuit switch channels.

If the TDM cycle controller generates time slot without considering empty time slots, no data are transferred through the network during some slots and network bandwidth is wasted. In this context, we propose the SES (skip empty slots) scheme to solve the problem.

Each register array that stores network configuration for time slot i generates a one bit signal, E_i, to indicate if there exists configurations for time slot i. The TDM cycle generator (see in Chapter 5) detects the E_i signal. The TDM cycle controller will skip the empty time slot and does not allocate time for it.

With SES, the network bandwidth utilization improves as shown in Figure 4.5(c). The TDM cycle controller skips the two empty slots and and slots are repeated in the sequence of $S_P^{(0)}$, $S_P^{(1)}$, $S_P^{(0)}$, $S_P^{(1)}$ and so on.

The TDM cycle controller with SES capability is based on the basic architecture, as shown in Figure 4.6. Signal E_i indicates if there are configuration assigned at time slot i. If the time slot i is empty, E_i equals 0. Signal $SLOT$ represents the number for current slot. $SLOT$ selects the one bit from E_0 to $E_K - 1$. Either it is time for the signal $TIME$ to toggle or the signal $EMPTY$ is '0', the signal $TRIG$ will switch its current value. Thus, the value of $SLOT$ is assigned to $SLOT\_PRE$ and starts computing the next slot number. Let us assume time slot j is empty, therefore E_j equals '0'. If $SLOT$ equals j, signal $EMPTY$ is '1'. The $TRIG$ will switch to its opposite value. At the same time, the COUNTER is reset. $SLOT\_PRE$ is equal to j, and current $SLOT$ equals time slot $(j+1) mod K$.

If a time slot does not contain network configurations, the time slot should not be counted in a TDM cycle.

4.4 SLA: SLOT LENGTH ADJUSTMENT

In SES, empty slots are skipped without occupying virtual channels. However, the length of time slots is equal. The ratio of the wormhole switched traffic and the predictive circuit switched may vary in different applications and in different phases of one application.

The fixed slot length introduces another kind of bandwidth waste. Each slot may contain valid network configurations. However a slot may not be fully utilized due to small amount of messages. Figure 4.7 shows an example of this scenario. Assume that the length of one slot is one time unit and that a processing element generates four chunks of data for predictive circuit switching and six packets for wormhole switching. In the example, the length of a

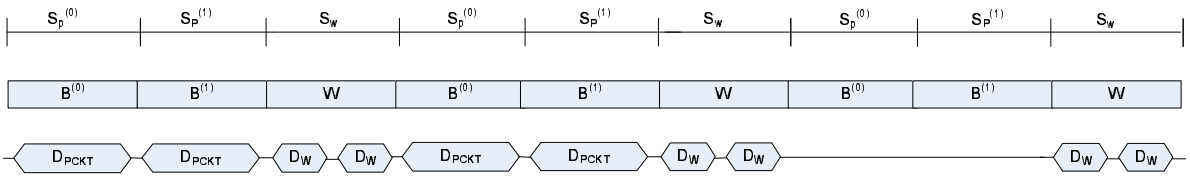
wormhole slot is the same as the length of a predictive circuit slot. During each slot, two packets or one chunk of data can be transferred. Figure 4.7(a) shows the data transmission through the network with equal length of slots. After two cycles, four chunks of data and four packets have been transferred. The rest of the two packets requires another cycle for transmission. Three cycles, or nine time units, are used for data transmission. In the system shown in Figure 4.7(b), the length of wormhole slot is 50% longer than the predictive circuit slot. Under this situation, two chunks of data can be transferred during one predictive circuit slot or three packets can be during one wormhole slot. Each cycle is 3.5 time units instead of three time units in previous case. It can be seen that after two cycles four chunks of data and six packets are transferred. It costs a total of seven time units and saves two time units compared with the example in Figure 4.7(a).

It can be concluded that the length of virtual channels assigned to predictive circuit switched traffics and wormhole switched traffics should comparable to their real traffic requirements.

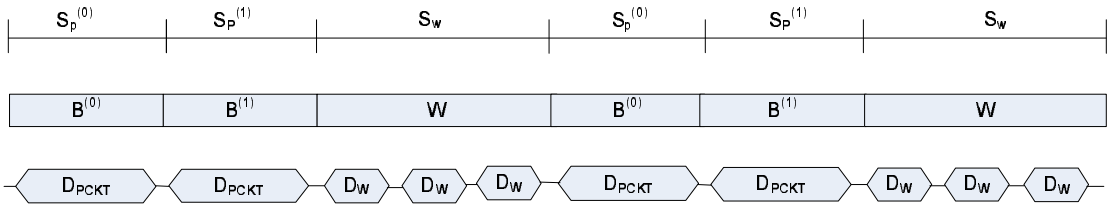
The virtual channels should be assigned according to the real traffic requirements. We modify the TDM cycle controller to enable the SLA capability. The modified architecture is shown in Figure 4.8. One more latch is added to the design. The values stored in the two latches, *NUM_P* and *NUM_W*, decides the length of predictive circuit slot and wormhole slot respectively. If the current slot number is K, the MUX selects the number stored for the length of wormhole slot. Otherwise, the number stored for the length of predictive circuit slot is selected.

4.5 PREEMPT: PREEMPT VIRTUAL CHANNELS

We propose preempt virtual channels (PREEMPT) method to solve the problem where has been shown in Figure 4.7(a). As apposed to SLA, PREEMPT does not modify channel assignment by adjusting the slot length. Since the traffic ratio of wormhole switched traffic and predictive circuit switched traffic may be unknown, PREEMPT handles the issue in a passive way. The mechanism is that if there exist predictive circuit switched traffic, the



(a) Time slots with equal length.



(b) Time slots with different length.

Figure 4.7: Network bandwidth utilization gets improved with SLA.

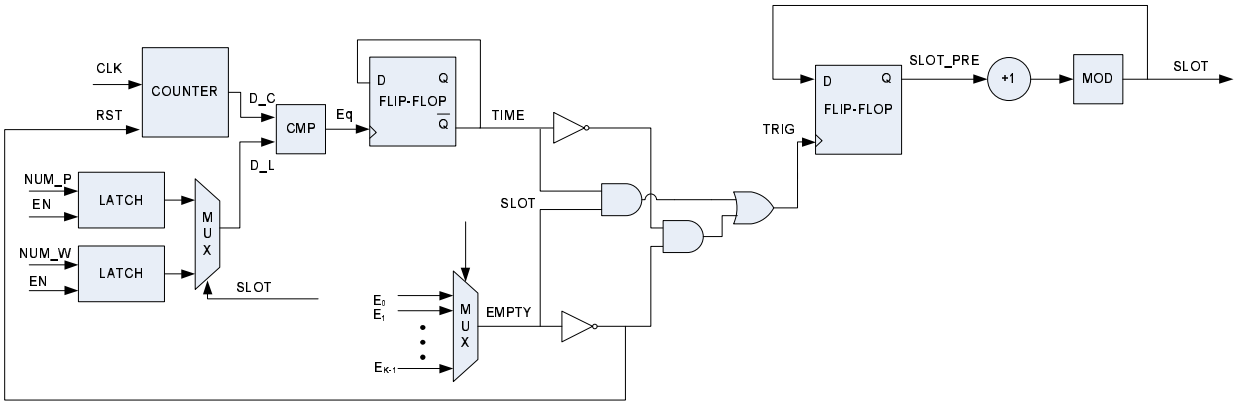
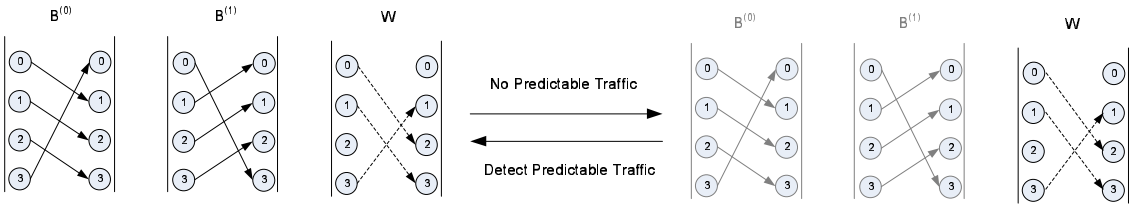


Figure 4.8: TDM cycle controller with SLA capability.

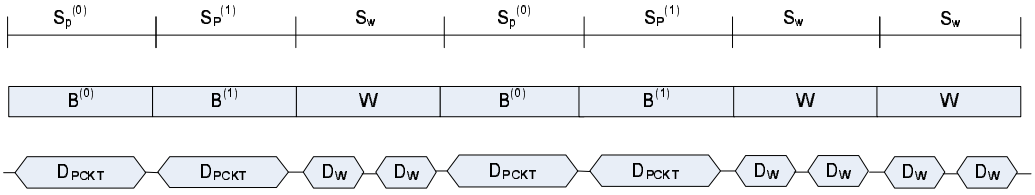
predictive circuit switched slots are counted. If not, the slots will be skipped. Figure 4.9(a) illustrates the mechanism of the approach. The configurations of $B^{(0)}$ and $B^{(1)}$ are sent to background if no predictive traffic is detected. After been sent to the background, the slot is used for wormhole traffic only. If predictable traffics are detected, then the configurations of $B^{(0)}$ and $B^{(1)}$ are re-activated. The time slots are counted for both predictive circuit slots and wormhole slots again.

We use the same example shown in Figure 4.7(a). Assume the slots have equal length. Two chunks of data or two packets are transferred during each slot. After one cycle, two chucks of data and two packets are transferred. Since there exist predictive traffics, the predictive slots are kept active. During the second cycle, two chucks of data and two packets are transferred. After that, all predictive circuit switched traffics are gone. So predictive configurations are sent to the background. The third cycle contains one wormhole slot. During that cycle, two packets are transferred. Because there are wormhole packets to be transferred, one more cycle is required for data transmission. Similarly, the fourth cycle contains one wormhole slot and the last two packets are transferred. It costs 7 time units totally, which is the same as that in SLA, but the network does not care about the traffic ratio information.

The PREEMPT is a good solution for traffic divided into phases. We have shown in Chapter 5 that if the phase are clearly known, we are able to pre-load configurations for different phases to achieve good network performance. PREEMPT gives a solution when the communication phase is not accurately predictable. The predictive circuit connections can be established for the communication pattern in the predictable phases. Those configurations are sent to background without consuming network bandwidth if the current communications do not match predictive communication patterns. When the application arrives at the particular communication phase, the predictive circuit connections wake up and make the network configured for data transmission. When the phase is complete, the network revert to wormhole switching. No matter how many times a certain phase appears, the network can be configured efficiently. In this way, the predictive circuit connections are used only when necessary.



(a) Network configuration with PREEMPT.



(b) Time slot assignment with PREEMPT.

Figure 4.9: Network bandwidth utilization gets improved with PREEMPT.

The architecture of TDM cycle controller with PREEMPT capability is shown in Figure 4.10. Predictive traffic indicates the TDM controller with a signal bit *Rc_v* telling if there exist predictable traffic. The *R_valid* is combined with E_i to decide if the slot i should be skipped. If no predictive traffics exist, *R_valid* is '0'. Then the signals input to MUX all equal '0'. In this way, the predictive circuit slots are skipped.

4.6 PERFORMANCE EVALUATION

We use 64 × 64 multi-processor model to evaluate network performance. The simulations discussed in this section focus on traffic, part of which is predictable. Data generation is 1 word in 2 ns. The scheduler's working frequency is set to 50 MHz. The wire in the network is operating at 6.4 Gb/s. A 64 × 64 crossbar fabric is used. The default length of one time slot is 200ns. Our simulation include (1) mixed random traffic and ordered-mesh traffic, and (2)traffic with unknown communication phase boundaries.

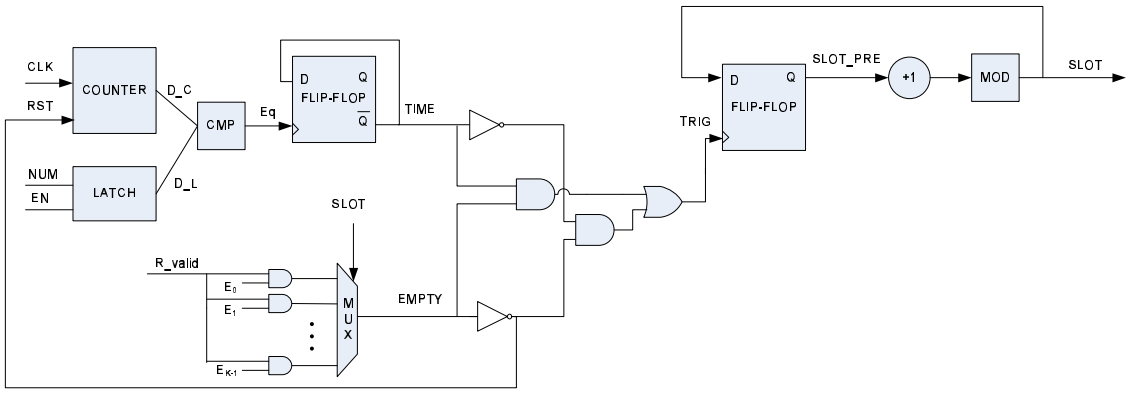


Figure 4.10: TDM cycle controller with PREEMPT capability.

4.6.1 Mixed traffics

We perform experiments on mixed traffics. The mixed traffic is generated by mixing random traffic with ordered-mesh traffic. The *predictable traffic ratio* is defined as the data that belongs ordered-mesh traffics over the total amount of data to be transferred. Traffic of various predictable traffic ratios are tested. SES is set as the baseline of our simulation results. The network throughput generated by pure wormhole switch, SES, and PREEMPT are compared.

Figure 4.11, 4.12 and 4.13 show the simulation on the hybrid switch using large, medium and small sized buffers within the NIC. The definition of the buffer size is based on the size of messages. In our simulation, the message size is set to 128 bytes. A large sized buffer is able to store 64 messages to avoid head-of-line completely. A medium size buffer is able to store 16 messages, thus messages may be blocked. A small sized buffer is able to store only one message, therefore messages are blocked.

We observed that SES performs better than wormhole switching when the predictable traffic ratio is higher than 80%. However, when the predictable traffic ratio is low, bandwidth utilization decreases.

Using large sized buffers, PREEMPT outperforms other schemes. This is because the buffers store a large amount of ordered-mesh traffic. In this way, during the time slot assigned for predictive circuit switch, the network bandwidth are fully utilized. After the ordered-mesh traffic is completely transferred, the predictive circuit connections are sent to background and the network is in pure wormhole switching mode. When the predictable traffic ratio is higher than 50%, SLA and PREEMPT provides better performance than wormhole switching.

Small to medium sized buffers introduce message blocking therefore the bandwidth assigned for predictive circuit switching is not always fully utilized. Using medium sized buffers, SLA and PREEMPT have similar performance and both exceed the performance of wormhole switching when predictable traffic ratio is higher than 70%.

Using small sized buffers, SLA and PREEMPT have performance gains when the predictable traffic is higher than 80%. When the predictable traffic ratio is lower than 80%. SLA still provides similar performance to wormhole switching. However, the performance of PREEMPT drops drastically. In mixed traffic the predictable traffic is spread out and predictable traffic appears occasionally during the whole communication. The possibility to preempt predictive circuit connections is low. In this situation, PREEMPT rarely preempts schemes and thus performs like SES.

4.6.2 Unknown phases

The traffics used in the experiments done for this section contain communication phases. We also use random traffic and ordered mesh traffic as two basic traffic patterns. Different from the previous section, random traffic is not mixed with ordered mesh traffic, but in its own phase. During one phase, ordered mesh traffic exists and during different phase, random traffic exists. We use the phase traffic to evaluate the capability of the PREEMPT method. The network throughput generated by SES, wormhole switch, SLA, and PREEMPT are compared.

We also simulated the network with large, medium and small sized buffers. The simulation results are shown in Figure 4.14, 4.15, and 4.16. As we expected, for all the experiments,

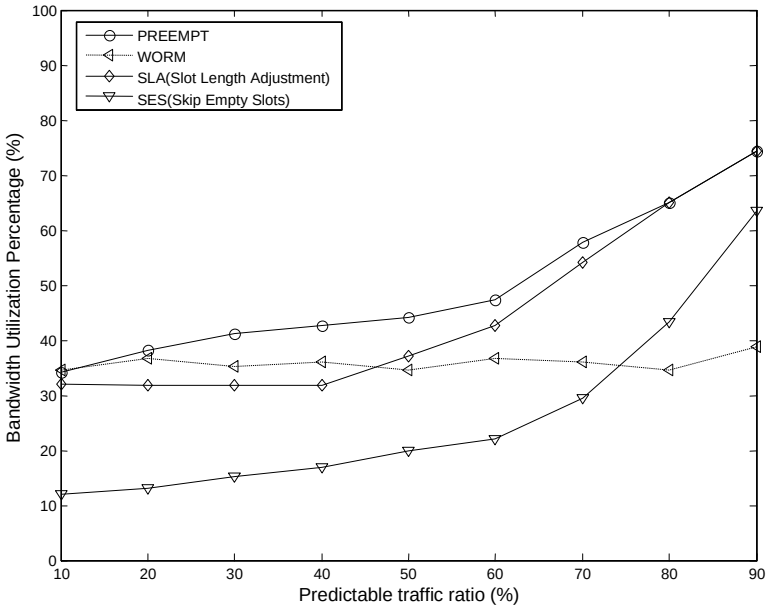


Figure 4.11: Mixed traffic (buffer size = 8 K bytes, message size = 128 bytes).

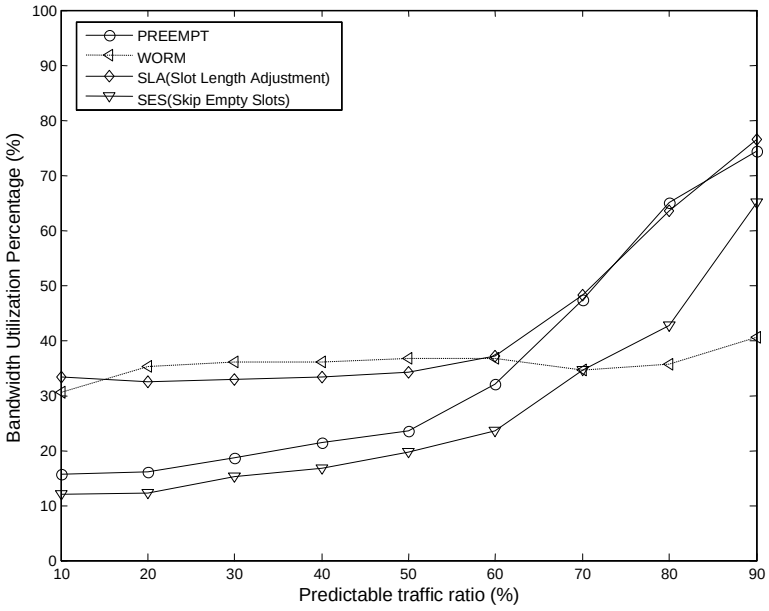


Figure 4.12: Mixed traffic (buffer size = 2 K bytes, message size = 128 bytes).

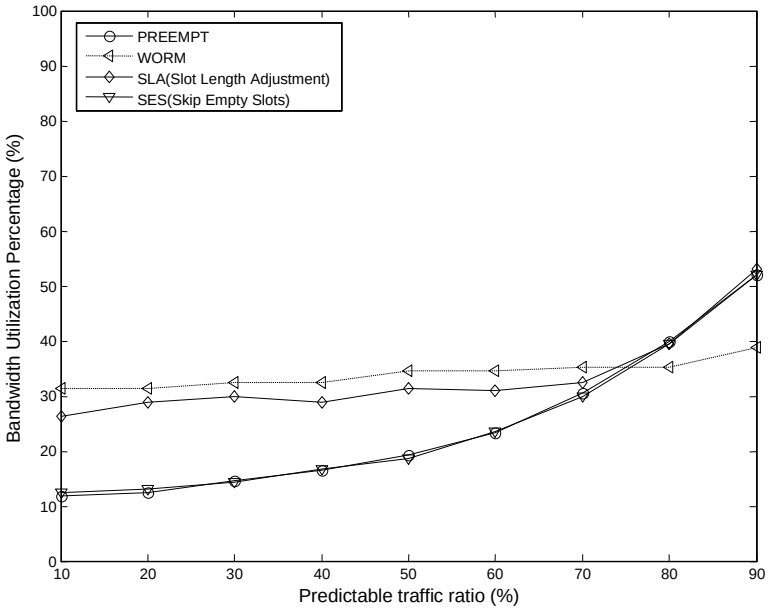


Figure 4.13: Mixed traffic (buffer size = 128 bytes, message size = 128 bytes).

the PREEMPT scheme provides the best performance as long as the predictable traffic ratio is no less than 10%. Because the predictable traffic is concentrated, the network bandwidth is highly utilized when the predictive circuit connection are activated.

In the network with medium sized buffers, SES and SLA provides better performance when the predictable traffic ratio is higher than 80%.

Figure 4.16 shows the performance of the network using small sized buffers. SLA and SES do not improve the bandwidth utilization. In the ordered-mesh communication phase, bandwidth assigned for wormhole is not used. In the random communication phase, the bandwidth assigned for predictive circuit switching is wasted. So SLA and SES perform even worse than pure wormhole switching.

The performance of wormhole switching is better when using small sized buffers than when using medium or large sized buffers. Using small sized buffers, messages are blocked if the previous message is not sent. During the ordered-mesh communication phase, messages are forced to be sent following ordered-mesh pattern. This helps to resolve communication conflicts in wormhole switching. Hence, the performance of wormhole switching is improved.

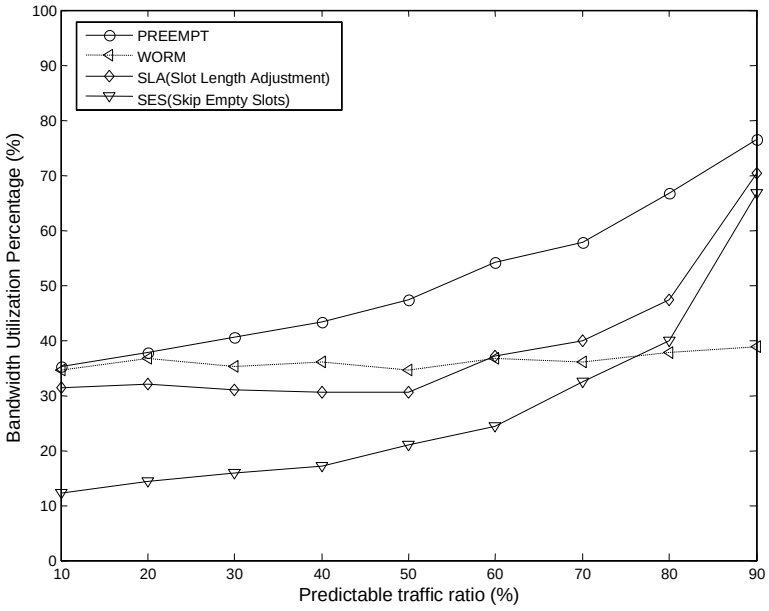


Figure 4.14: Unknown phases (buffer size = 8 K bytes, message size = 128 bytes).

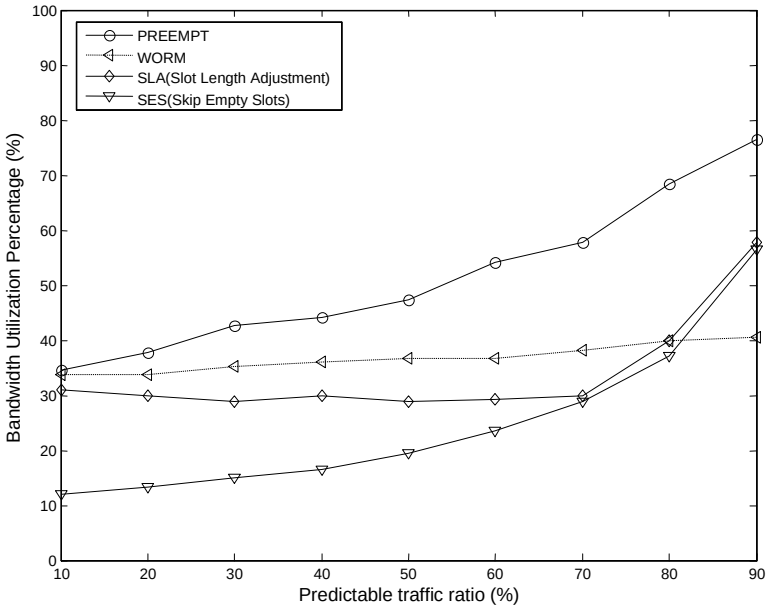


Figure 4.15: Unknown phases (buffer size = 2 K bytes, message size = 128 bytes).

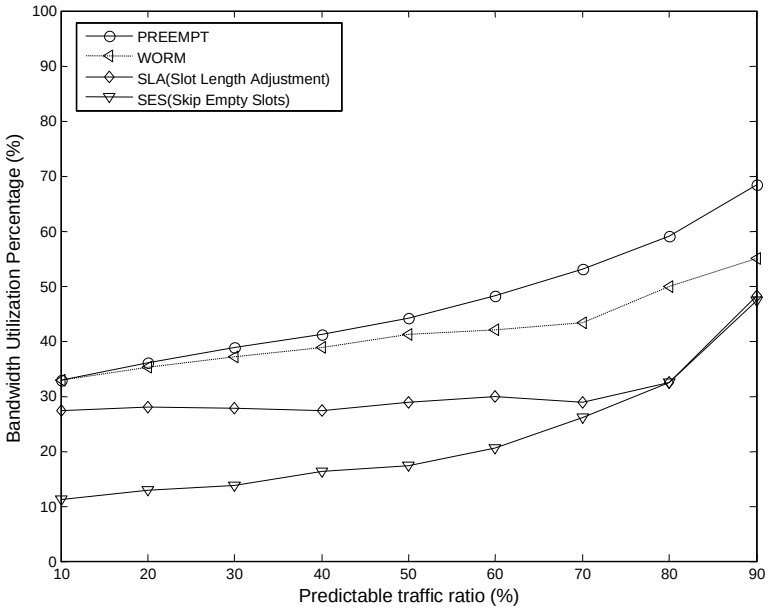


Figure 4.16: Unknown phases (buffer size = 128 bytes, message size = 128 bytes).

4.7 CONCLUSION

This chapter discussed virtual channels assignment issues. Two observations are made that network bandwidth are wasted mainly when (1) time slots are not filled with network configurations and when (2) network configurations and network traffics does not match. Based on the observations, we propose three virtual channel assign schemes to improve network performance, which is SES, SLA, PREEMPT. Our experiments show that PREEMPT provides best performance without clear knowledge of traffic ratio or communication phase boundaries as long as the NIC buffers are large.

5.0 REAL-TIME GREEDY SCHEDULER

In this chapter, we present a real-time greedy scheduler for an NxN crossbar interconnection system for predictive circuit switching. The main approach to achieve better network performance is by amortizing the control overhead of connection establishment, as described in Section 5.1. Three mechanisms are proposed in Section 5.2 for the predictive control of networks. Section 5.3 gives detailed hardware architecture of the real-time greedy scheduler. Hardware performance is evaluated in Section 5.4. The network performance using circuit switching, packet switching and the predictive circuit switching is compared in Section 5.5. Our result shows that with our real-time greedy scheduler, the predictive circuit switching provides better performance than circuit switching and wormhole switching under predictable traffic.

5.1 AMORTIZING THE CONTROL OVERHEAD OF CONNECTION ESTABLISHMENT

A dedicated circuit between a source and a destination is very effective for data transmission since it simplifies tremendously the communication protocols. Specifically, no congestion control is needed, no routing or control information has to be included with the data, no intermediate buffering and routing is needed and only end-to-end flow control is required. However, circuit switching is only effective when the cost of establishing the circuits is small compared to the cost of transmitting the data, which is only the case when an established circuit is extensively used before it is torn down.

Circuit switching would be an ideal switching scheme if the capacity of the interconnection network would be large enough to satisfy all the connections requested during execution without any conflict. For instance, if C is the set of connections that are used during the execution of a parallel program and the interconnection network can satisfy all the connections in C without conflict, then circuit switching will pay the overhead of establishing each connection only once during the execution of the program. Specifically, the overhead will be paid only when a connection is requested for the first time. Moreover, if C is known before the program starts execution, then the network can be configured to satisfy the connections in C before execution starts, thus removing any run-time overhead to establish the circuits.

Unfortunately, a scalable interconnection network cannot realize all the communication requirements of real applications without conflict. Due to the limited capacity of networks, circuits will have to be repeatedly torn down and re-established during execution. Thus, the resulting overhead may cause a severe bottleneck at the scheduler which will have to resolve contention among competing requests and establish new circuits at run time.

A possible solution to the problem of limited network capacity is to decompose the set of connections, C, into a number of subsets, C_1 ,..., C_k, such that $C = C_1 \cup ... \cup C_k$, and each C_i, $i = 1, ..., k$, can be realized in the network without conflict [44]. Time division multiplexing (TDM) can then be used to realize each set C_i periodically in a separate time slot. We call any set of connections that can be realized in the network without conflict, a network configuration set, or simply a configuration. Hence, with TDM, the connections in each configuration set C_i could be realized in the network every k time slots.

Although using TDM all the connections in C can coexist in the network, each connection gets only *1/k* of the maximum possible connection bandwidth. Hence, it is imperative to keep k as small as possible. Exploring communication locality can be very useful in that regard since it implies that only a subset of C is being used at any given time during the execution of a program.

Specifically, assume that $W^{(1)}, ..., W^{(p)}$, is a sequence of sets of connections that represents the communication requirements during the execution of the program. That is, the program goes through p phases and during the execution of each phase, $j, j = 1, ..., p$, it uses the connections in set $W^{(j)}$. We call each set $W^{(j)}$ a communication working set. Note that the

sets $W^{(1)}, ..., W^{(p)}$ are not necessarily disjoint, but $W^{(1)} \cup ... \cup W^{(p)} = C$. During the execution of phase j, the multiplexing degree is set to k_j, where the set $W^{(j)}$ can be decomposed into k_j network configurations.

The partitioning of the communication requirements into phases is not unique, but is strongly influenced by the communication locality. For programs with strong spacial communication locality, it should be possible to find a partitioning in which the size of each working set $W^{(j)}$ is small, thus leading to a small multiplexing degree k_j. For programs with strong temporal communication locality, the number of phases, p, should be small leading to fewer network reconfigurations during execution. This is a desirable property since network reconfiguration and circuit establishment will be performed at the rate of the change in communication locality, rather than at the rate of communication requests.

In Section 5.2, we will discuss different schemes for identifying communication phases present during program execution. However, it should be clear that there is a tradeoff between the number of phases, p, and the size of each working set $W^{(j)}$. At one extreme, we can consider that $p = 1$ and that the entire execution is regarded as a single phase, with $W^{(1)} = C$, and $k_1 = k$. At the other extreme, p is considered to be as large as it takes to allow the connections in each working set $W^{(j)}$ to be realizable in the network without conflict. More phases lead to more frequent reconfigurations and thus to larger reconfiguration overhead while larger than necessary multiplexing degree leads to inefficient network utilization. Specifically, if during a phase j, the actual communication traffic utilizes only s of the k_j multiplexed slots needed to establish the working set, then only s/k_j of the available network bandwidth is utilized.

5.2 PREDICTIVE CONTROL OF NETWORKS

Traditional circuit switching falls naturally into the general framework described in Section 5.1. Specifically, circuit switching amounts to TDM with a multiplexing degree of one. Hence, each realizable active working set is necessarily a configuration that can be established in the network without conflict. Moreover, the establishment of each new requested circuit

represents a change in the active working set. This change may require removing some existing connections even if these connections must be re-established in the near future. As discussed in Section 5.1, it is crucial for communication efficiency to track and minimize the active working of the running application. TDM allows caching larger working sets of connections, and provides the ability to change the multiplexing degree as required by the application. In the following, we explore different schemes for identifying, predicting, and tracking communication working sets.

5.2.1 Compile-time and load-time prediction of working sets

Many parallel applications have regular communication patterns that can be identified either at compile time or at load time after the mapping of the application to processors is determined. In [25], an experimental compiler was developed to determine the communication requirements of programs written in a shared memory language, such as OpenMP. A similar concept was applied in [45] to thread level computations and in [24] to programs that use message passing. In general, it was found that in parallel scientific applications, most inter-processor communications can be determined at compile or load time [46, 47].

Developing parallel programs using message passing gives application developers explicit control over inter-processor communications, while many shared memory parallel languages give the application developer explicit control over the allocation of the address space to memory modules. Moreover, new languages such as StreamIt [48], assume that communication patterns between processes are specified and MPI has facilities called *communicators* for explicitly specifying the communication working set. In order to obtain efficient programs, users usually take advantage of the capability to control communications and memory allocation. Hence, it is reasonable to ask the user to give some directives to the compiler about the active communication working set in different phases of the program, if the user wishes to increase the efficiency of inter-processor communication.

Compiled communication allows the compiler to statically determine and optimize the communication requirements in parallel systems [49]. It has been used in combination with message passing in the iWarp system [50, 51]. In [52], the compiler inserts the commands

needed to establish the needed connections in the network before the communication takes place. However, because circuit switching is used, the overhead of establishing the connection turned out to be extremely large. In our work, we use TDM to take advantage of compiled communication for static communication patterns without the significant overhead of circuit switching. A similar TDM approach is proposed in [45] for adaptive System-On-a-Chip.

In this dissertation we will not elaborate on compiler technology, but we will assume that the compiler can identify the appropriate communication working sets when such an identification is possible [45, 53]. Instead, we will present in Section 5.5 the design of a communication network which can greatly benefit from compiler identification of the communication patterns.

5.2.2 Dynamic prediction of the working set

Branch prediction has been proved to be a very powerful technique for improving the performance of microarchitectures, and many attempts have been made to apply the same concepts to improve communication performance. The idea is to predict the communication requirement and to establish the corresponding circuits in the network before they are actually needed, thus eliminating circuit establishment overhead. Using the notation from Section 5.1, the works in [54, 55], for example, attempt to predict the connections in the working set $W^{(j+1)}$ while $W^{(j)}$ is being used. In order for such prediction to be useful, however, the processors should be doing useful computational work while the network is being configured from $W^{(j)}$ to $W^{(j+1)}$.

When TDM is used to increase the size of the set of established connections, the overhead of adding a new connection is incurred only the first time the connection is used. Once a connection has been established in the network, there is no overhead for reusing the connection. The overhead of establishing a connection when it is used for the first time is similar to the penalty for compulsory misses in caches; if the right cache size is used, then a cache miss occurs only on the first reference to a memory location, while successive references to the same location are all hits. In order to keep the multiplexing degree small, however, a connection which will no longer be used should be removed from the working set. Going

back to the cache analogy, trying to keep the multiplexing degrees small is similar to allowing the cache size to decrease by evicting cache lines before they have to be replaced with other cache lines.

Hence, instead of trying to predict when to add a new connection to the working set, the role of dynamic predictions in our network is to predict when to remove a connection from the working set. The purpose of this dissertation is not to compare the effectiveness of different predictors but to present a network architecture that will allow such prediction. For this reason, we will use in our experiments a simple 'time-out' predictor in which a connection is removed if it is not used for a certain period of time. A different predictor can be implemented by associating a counter with each connection in the working set. This counter is reset to zero every time that connection is used and is incremented every time another connection is used. When the counter reaches a certain threshold, the connection is evicted from the network. In other word, a connection is evicted if it is not used while other connections are being used, but is not evicted if the application is in a computation phase, where no communication takes place.

5.2.3 Dynamic reconfiguration with compiler assistance

High-level knowledge of the program's structure is useful to dynamic prediction discussed in Section 5.2.1. This information can either be provided by the user as directives or in many cases discovered by a compiler. For example, consider a compiler that detects different communication patterns between two consecutive loop structures. Even if the compiler cannot detect the patterns themselves, it can insert an instruction in the code that flushes all current connections in the network between the two loops. Thus, when the second loop executes it will not mis-predict the pattern based on the previous loop, but rather build a new working set immediately. This idea has been verified by the work in [53]. Other points that may indicate changes in communication localities include procedure boundaries, "if" statements, and points of remapping tasks to processors for load balancing.

The compiler can assist a dynamic reconfiguration strategy considerably in more subtle ways. The compiler might be able to statically determine a portion of the working set,

allowing the dynamic reconfiguration strategy to only work on non-predicted communications. For example, consider the case where a loop contains an embedded "if" statement. The communication pattern for the loop may now depend on the condition of "if" statement. The predictor's knowledge of the conditional can significantly simplify the communication pattern detection. One way this could be used is to store a second level working set that is swapped in only when the conditional is true.

If the compiler can predict only a portion of the communication operations statically, the predicted configurations can be preloaded to the network, while the scheduler can continue to schedule dynamically requested connections that are not preloaded.

5.3 HARDWARE ARCHITECTURE OF PREDICTIVE CIRCUIT SWITCHING SCHEDULER

Chapter 3 gives overall architecture of the hybrid switching system. The switching fabric in the system is a passive fabric with no buffering or control capabilities. The fabric can represent a crossbar interconnection, a multistage fabric, a fat tree organization, or any other direct interconnection topology.

5.3.1 Scheduler architecture

The configuration of the fabric is determined by configuration registers. By loading specific values into the registers, specific mappings between the input ports and the output ports are realized. In its simplest form, a configuration, C, may be represented by a Boolean matrix, B, where $B_{u,v}$ is '1' when input u is connected to output v, and $B_{u,v}$ is '0', otherwise. For the case of a crossbar fabric, the only constraints on B are that there is at most one non-zero entry in each row and at most one non-zero entry in each column. More complicated constraints may be derived for fabrics that have limited permutation capabilities (e.g. multistage networks) or multiple-paths from inputs to outputs (e.g. fat tree networks). In the remainder of this chapter, we will present a detailed design for a system based on a crossbar fabric.

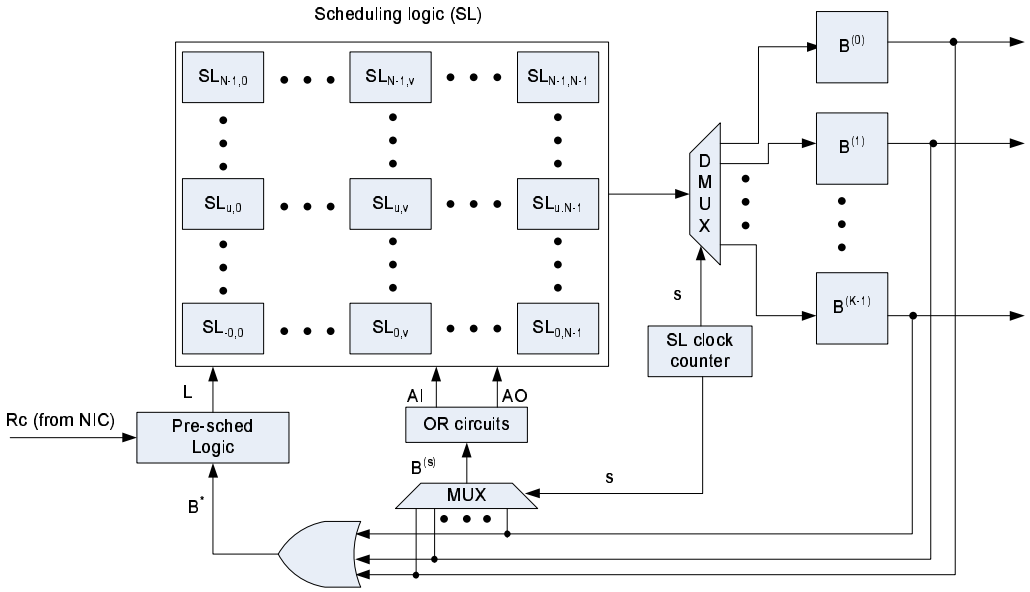


Figure 5.1: A detailed diagram of the scheduler.

The scheduler receives a request, $R_u, 0 \leq u \leq N-1$, from each of the N NICs indicating which of the logical queues of that NIC is not empty. Hence, each R_u is an N-bit signal, $R_{u,0}, ..., R_{u,N-1}$, transmitting to the controller the communication requirements of NIC$_u$. The controller receives requests for connections from all the NICs (a matrix R), schedules the connections and communicates an N-bit grant signal, G_u , to each NIC$_u$. The scheduler sets the v^{th} signal of G_u, $G_{u,v}$, to 1 whenever a circuit is established between the output port of NIC$_u$ and the input port of NIC$_v$. At most one of $G_{u,v}, v = 0, ..., N-1$ can be non-zero at any given time. Note that the grant signals $G_0, ..., G_{N-1}$, are the rows of the configuration matrix B.

In order to support a multiplexing degree of K, the scheduler has to create K configuration matrices, $B^{(0)}, ..., B^{(K-1)}$, one for each of the K multiplexed time slots. The scheduler satisfies requests from NICs in any of the K slots. However, in any given time slot, t, only the corresponding matrix $B^{(t)}$ is copied to the fabric configuration register and the corresponding grant signals are sent to the NICs. Figure 5.1 shows a detailed diagram of the

scheduler, in which a time slot clock controls the copying of the configuration matrix to the switch fabric. The TDM counter shown in the figure is a counter which counts from 0 to K-1, but which skips a particular count t, if the corresponding matrix $B^{(t)}$ is all zeros. This feature skips over empty configurations and allows the scheduler to reduce the multiplexing degrees by controlling the content of the configuration register.

Note that in the above design, we provide an explicit grant signal from the scheduler to the NICs, thus giving the scheduler the responsibility of controlling the synchronization among the NICs. Specifically, the grant signals indicate to each NIC the period in which it can send data, thus removing the need for the NICs to keep track of the TDM slot boundaries. However, a *guard band* should be enforced between consecutive time slots. During that band, circuits should not be used due to uncertainties in the fabric state. The length of the guard band depends on the variations of the propagation delays of the grant signals and on the time needed to change the setting of the switch fabric. For example, when 1 μs time slots are used, if the time to reconfigure the switch fabric is within 50 ns and the maximum length of a grant line is 50 feet (50 ns propagation delay), then the length of the guard band is 50 ns, which means that 5% of each time slot cannot be used for data transfer. Note that during a 1 μs slot, 125 bytes of data can be transmitted per serial Gb/s link.

5.3.2 Scheduling logic

The block designated "scheduling logic" in Figure 5.1 is responsible for generating the schedule for a particular time slot, s. The SL counter selects the time slot, s, $0 \leq s \leq K$, to which it will try to insert the pending requests. Assuming that the current multiplexing degree is k, $k \leq K$, a simple scheme to select s is to apply a round robin rotation among the k currently active configurations. The current configuration matrix for slot s, $B^{(s)}$, is selected by a multiplexer and fed to a pre-scheduling logic, along with the request matrix R and a matrix B^* which is set to $B^{(0)} + ... + B^{(K-1)}$,where $+$ is the bit-wise OR operation. The matrix B^* represents all the connections that are currently established in the network (in any of the K time slot). Specifically, $B^*_{u,v} = 1$ if and only if the connection from port u to port v is established during any one of the K time slots. By comparing $B^*_{u,v}$, $B^s_{u,v}$ and $R_{u,v}$,

Table 5.1: The possible inputs to the pre-scheduling logic

$R_{u,v}$	$B^{*}_{u,v}$	$B^{(s)}_{u,v}$	*Description of the case*	$L_{u,v}$
0	x	0	Connection not requested and not realized in slot s	0
0	x	1	Connection not requested and realized in slot s (should release)	1
1	1	x	Connection requested and realized in some slot	0
1	0	0	Connection requested and not realized in any slot (should establish)	1

Table 5.2: The function of a scheduling logic module, $\mathrm{SL}_{u,v}$

$L_{u,v}$	$A_{u,v}$	$D_{u,v}$	*Action*	$T_{u,v}$	$A_{u+1,v}$	$D_{u,v+1}$
0	x	x	No change in connection	0	$A_{u,v}$	$D_{u,v}$
1	1	1	Release the connection in slot s	$1(B^{(s)}_{u,v}1 -> 0)$	0	0
1	1	0	Need connection but resources not available	0	$A_{u,v}$	$D_{u,v}$
1	0	1	Need connection but resources not available	0	$A_{u,v}$	$D_{u,v}$
1	0	0	Establish connection in slot s	$1(B^{(s)}_{u,v}1 -> 0)$	1	1

the scheduler can figure out whether it needs to make any change to the value of $B^s_{u,v}$ (the state of the connection from u to v in slot s).

In Table 5.1, we describe the possible cases that the pre-scheduling logic has to deal with. The value of $L_{u,v}$ shown in the last column of the table is generated to be equal to 0 if no change is to be made in the value of $B^s_{u,v}$. The value of $L_{u,v}$ is equal to 1 either if a connection is to be released or if a connection is to be established. In order to release and establish connections, we need to keep track of resource availability. For crossbar switch fabrics, resources are output and input ports. Hence, two vectors AO and AI are used to express the availability of ports in the current schedule of slot s. Specifically, AO is obtained by taking the "or" of the columns of $B^{(s)}$. That is, $AO_v = B^{(s)}_{0,v} + ... + B^{(s)}_{N-1,v}$, which is equal to 0 if and only if output port v is unscheduled in slot s (no input is connected to output v). Similarly, the vector AI is obtained by taking the "or" of the rows of $B^{(s)}$. That is, $AI_u = B^{(s)}_{u,0} + ... + B^{(s)}_{u,N-1}$ which is equal to 0 if and only if input port u is unscheduled in slot s (input u is not connected to any output).

The scheduling logic for a crossbar switch is composed of an NxN array of identical modules, $\mathrm{SL}_{u,v}$, $u = 0, ..., N-1$, $v = 0, ..., N-1$. Each $\mathrm{SL}_{u,v}$ receives the signal $L_{u,v}$ and is responsible for scheduling or releasing the connection from input port u to output port v. Two sets of port availability signals propagate in the SL array to carry information about the availability of input and output ports in slot s. One set of signals, $A_{u,v}$, propagates upwards through rows $0, ..., N-1$ of the array and is initialized such that $A_{0,v} = AO_v$ for $v = 0, ..., N-1$. The other set of signals, $D_{u,v}$, propagates rightwards through columns $0, ..., N-1$ and is initialized such that $D_{u,0} = AI_u$ for $u = 0, ..., N-1$. At any given scheduling module, $\mathrm{SL}_{u,v}$, the input $A_{u,v}$ is equal to 0 if and only if output port v is available (not occupied) and $D_{u,v}$ is equal to 0 if and only if input port u is available (not occupied). Each $\mathrm{SL}_{u,v}$ passes $A_{u,v}$ upward (as $A_{u+1,v}$) and $D_{u,v}$ rightward (as $D_{u,v+1}$) unchanged if $L_{u,v} = 0$. However, if $L_{u,v} = 1$, then $\mathrm{SL}_{u,v}$ sets $A_{u+1,v} = D_{u,v+1} = 0$ if it is releasing the connection between ports u and v, or sets $A_{u+1,v} = D_{u,v+1} = 1$ if it is using input ports u and output port v to establish a connection.

Figure 5.2 and Table 5.2 describe the way the availability signals propagate in the scheduling array. Table 5.2 also specifies the output signal $T_{u,v}$ generated for each input combina-

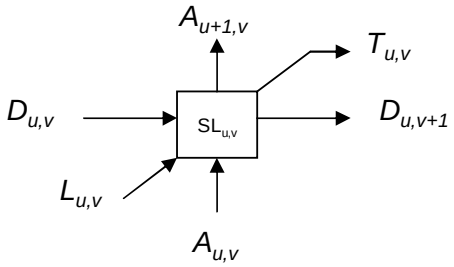


Figure 5.2: The inputs and outputs to $SL_{u,v}$.

tion. An output $T_{u,v} = 0$ means that the value of $B_{u,v}^{(s)}$ should not be changed, while an output $T_{u,v} = 1$ means that the value of $B_{u,v}^{(s)}$ should be toggled. Note that by initializing $A_{0,v} = AO_v$, $v = 0, ..., N-1$, and $D_{u,0} = AI_u$, $u = 0, ..., N-1$, we make the unused network ports available to a request $R_{u,v}$ before they are available to another request $R_{a,b}$ if $u < a$ or $v < b$, thus always giving a higher priority to the former. A more fair schedule can be obtained by rotating the priority such that $A_{a,v} = AO_v$, $v = 0, ..., N-1$, and $D_{u,b} = AI_u$, $u = 0, ..., N-1$, where a and b are selected randomly or through a round robin scheme.

The output of the scheduling logic, T ($T_{u,v}$, $u, v = 0, ..., N-1$) is then used to update the configuration matrix $B^{(s)}$ (see Figure 5.1), thus completing the scheduling process for slot s in one SL clock cycle. Note that the period of the SL clock depends on the propagation delay in the scheduling logic and is independent of the period of the time slot clock. In other words, the scheduling for a particular slot, s, is performed while the switch fabric is configured according to the configuration for a possibly different time slot t.

5.4 HARDWARE PERFORMANCE

Because of the time needed for the signals $A_{u,v}$ and $D_{u,v}$ to propagate in the SL array, the scheduling delay should be linearly proportional to the system size, N. We have synthesized the scheduler circuit on an Altera Stratix FPGA (EP1S25F1020C-5), and the latency of

the resulting circuit is shown in Table 5.3 for different system sizes. We observe that the scheduler is salable. The number of scheduled connections per second increases as the system gets large. ASIC results tend to be 5 to 10 times better than the FPGA results. In the simulation described in Section 5.5, we conservatively chose the ASIC performance to be 80 ns for a 128x128 scheduler (about 5x better).

Table 5.3: Latency of the scheduling circuit

System size	4	8	16	32	64	128
Latency (ns)	34	49	76	120	213	385
Millions of Possible Scheduled Connections /second	118	167	211	266	300	332

5.5 SYSTEM EVALUATION

For our simulations, we created a multi-processor model that contains a single crossbar for communications and a single scheduler for arbitration. Other interconnection fabrics are possible but this represents a baseline topology. We have simulated a 128 processor system that supports wormhole routing, circuit switching, and multiplexing of the communication pattern with dynamic scheduling and preloading a set of communication patterns. Predictive communications utilize the ability to preload the communication pattern into the network. When prediction is not possible, or in cases of misprediction, dynamic scheduling can be employed.

5.5.1 Network simulation methodology

The simulations are performed on a cycle-accurate simulation framework. The brief simulation methodology is described in this section to provide background knowledge about network performance evaluation. If interested, please refer to Chapter 8 for detailed information.

The network simulator is built as a common framework for designing, synthesis, and simulating parallel computing network. We are able to create networks with different switching technology using modular components. Each components can be designed separately and characterized in terms of latency and bandwidth. The latency is represented as multiple cycle latency. The parameters, e.g. cycle and latency, of each components are provided by hardware synthesis tools. In this way, we can accurately determine the latency down to nanosecond (10^{-9}) level of accuracy. The components built for network simulation include processing elements (PE), network interface cards (NIC), wires, network schedulers, and switch fabrics.

By using the VHDL hardware description language to construct a hardware simulator, we are able to simulate the entire network to cycle accuracy. This level of simulations enables us to observe the true behavior of the network with specific switching techniques. The hardware behavior and performance is then used to create identical modules in SystemC. SystemC is C++ based system design language that was developed for component to system level simulations. On one hand, SystemC has inherited flexibility from C++ so that the simulator development cycle is decreased. On the other hand, the SystemC allows to create structure that are available in hardware description language, such as bit-vetors, ports, processes. The network components and systems built in SystemC are validated with those built in VHDL to guarantee identical functionality. Using SystemC simulator, we are able to simulate networks with 128+ communication nodes.

5.5.2 Simulation result

Each of the 128 processors is modeled as a packet generator/receiver and contains a command file that defines the type and sequence of communications. The network interface card(NIC) was designed using synthesizable VHDL and requires a single-cycle delay of 10ns to send or receive data. This performance is optimized but represents real hardware that has been synthesized. The wires in the network assume 10 foot cables using high-speed serial links operating at 6.4 Gb/s. The latency is modeled as a 30ns delay for parallel-to-serial conversion, 20ns for propagation delay down a ten foot wire and 30ns for serial-to-parallel conversion.

For all networks, a 128x128 crossbar fabric is used. For the wormhole routed switch, the crossbar is digital but for the other networks, the crossbar is a Low-Voltage Differential Signal (LVDS) switch. The propagation delay through the digital switch is modeled as 10 ns while the propagation delay through the LVDS switch is neglected as it requires less than 2 ns (equivalent to a 1 foot cable) [16]. An 80 ns scheduler is used for all network types, as described in Section 5.3.

For a wormhole message, the delay through the switch includes the time required to schedule the first flit of the message, which is 80 ns. All subsequent flits in the same worms are routed in 10 ns. In order to ensure fairness within the network, worm sizes are limited and in our simulation we set this limit to 128 bytes. The flit size is 8 bytes. It should be noted that if a message is broken up into two worms, the cable delay is only seen once as the second worm is buffered within the crossbar switch.

For circuit switching, however, the delay to schedule a message includes the cable delay of 80 ns to send the request, 80 ns to schedule the request, and another 80 ns to send the grant back to the NIC. After that, the point-to-point delay is 30+20+20+30 ns.

We ran four test patterns using message sizes from 8 to 2048 bytes: Scatter, Random Mesh, Ordered Mesh, Two Phase, and Hybrid. These patterns were selected based on a study of the NAS benchmarks [20].

5.5.2.1 Preloading The Scatter test sends a unique message from a single processor to all 128 processors. Random Mesh represents nearest neighbor communications in a 2D mesh but without any predictability while Ordered Mesh represents an ordered nearest neighbor communication pattern. The Two Phase test represents those programs that contain global communication and local communication. In this test, there is one 128-processor all-to-all communication followed by 16 random nearest neighbor communications.

The simulation results are shown in Figure 5.3. For the Scatter test pattern, there is a notable increase in bandwidth utilization between 32 and 64 bytes. This is due to the fixed duration of each of the communication cycles. Each cycle is fixed at 100 ns or 80 bytes.

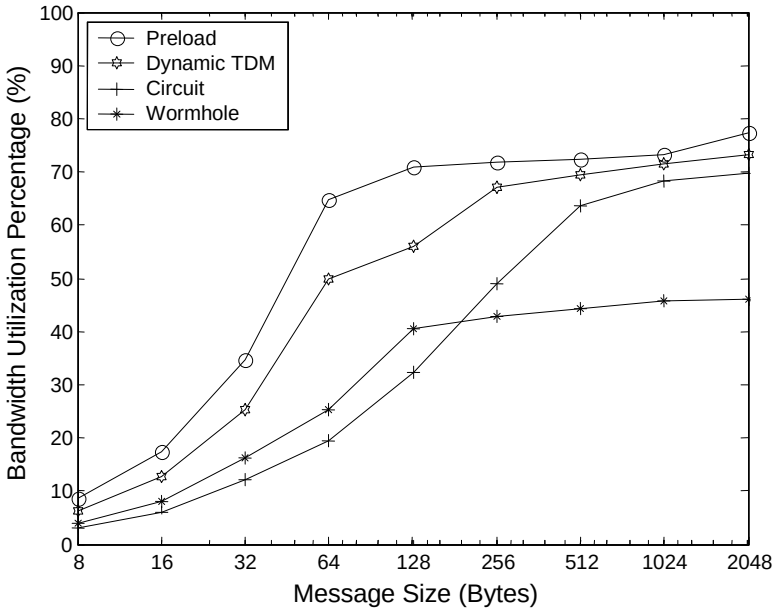
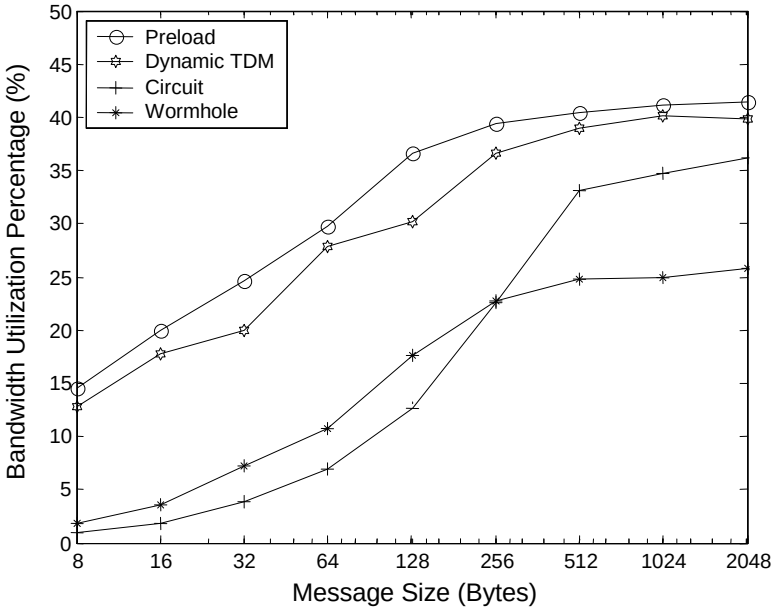


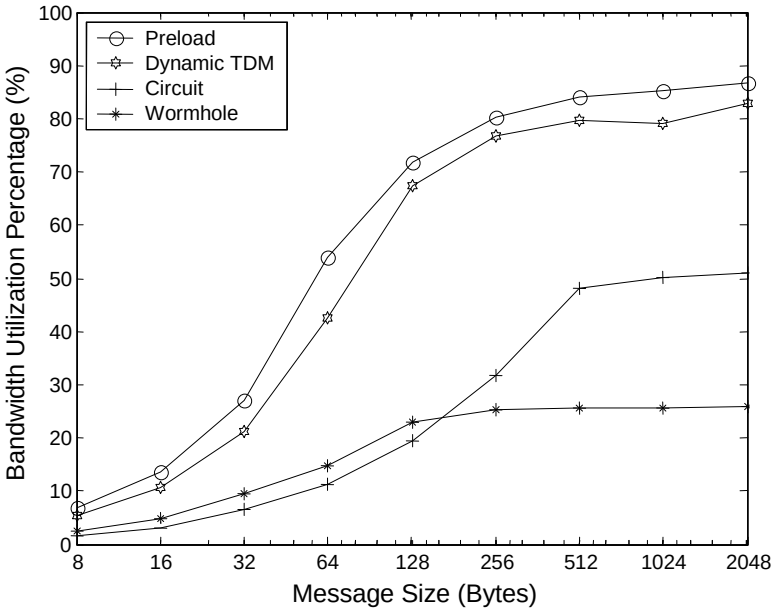
Figure 5.3: Performance results for scatter. The Preload and Dynamic TDM utilize a multiplexing degree of four.

Messages between 8 and 64 bytes can be transmitted in a single cycle. Messages over 80 bytes are fragmented into multiple cycles and must remain idle when its communication cycle is not active. This is why the efficiency flattens out form 64 to 2048 bytes.

For Preload versus Dynamic TDM, it can be seen that the Scatter performance is very similar. For Random Mesh, both Preload and Dynamic TDM outperform Wormhole and Circuit switching by 10% to 25% but are within 10% of each other, as shown in Figure 5.4. The performance of Circuit switching improves as the message size increases. The Ordered Mesh pattern represents communications that are highly predictable. In our experiments, 4 destinations were used per processor and thus, there was still a relatively high hit-rate for dynamic scheduling of TDM. The Ordered Mesh, as one would expect does very well with Preload. The regularity of the pattern also shows good efficiency for TDM but is not exploited for Wormhole or Circuit switching. For a larger number of destinations, the efficiency of dynamically scheduling TDM is expected to decrease.



(a) Random mesh



(b) Ordered mesh

Figure 5.4: Performance results for random mesh and ordered mesh. The Preload and Dynamic TDM utilize a multiplexing degree of four. Ordered and random mesh represents nearest neighbor communications for a 2D mesh.

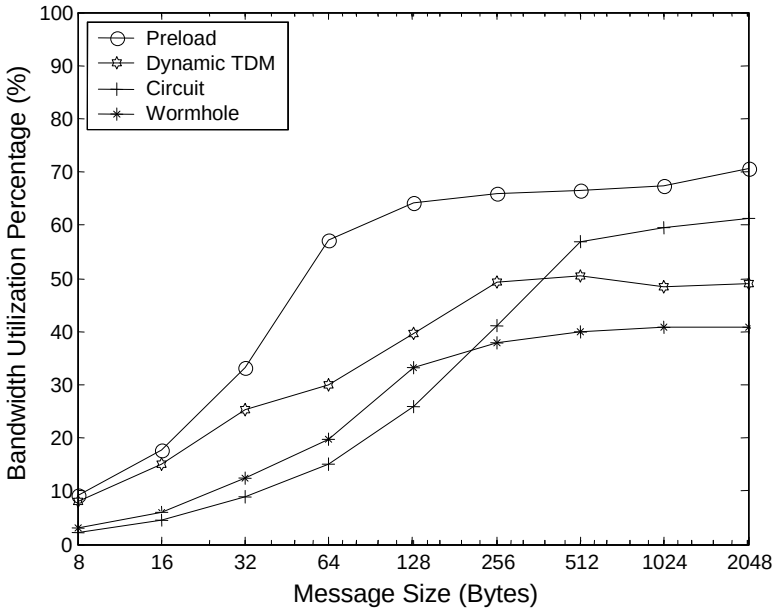


Figure 5.5: Performance results for two phases. The Preload and Dynamic TDM utilize a multiplexing degree of four.

5.5.2.2 Setting phases For the Two Phased communication test as shown in Figure 5.5, Preload does better than the rest and the performance of dynamically scheduled TDM drops below Wormhole. This is due to the fairly small set of destinations in the Random Mesh phase and due to the highly structured nature of the All-to-All phase. For the Random Mesh phase, we have shown that both Preload and Dynamic TDM do well but the All-to-All pattern is only exploited by preloading.

5.5.2.3 Partial preloading We also simulated the capability of the switch to deal with dynamic communications while preloading the statically known communication patterns. For this experiment, a percentage of the communications are to specific processors and the remaining are randomly sent to any processor. We select a multiplexing degree and we use k slots to preload the static patterns, while the other 3-k slots are use to schedule dynamic communication. We changed k between 0 and 2, and the results are shown in Figure 5.6. The 1-preload/2-dynamic outperforms the pure dynamic scheme even for low determinism (50%).

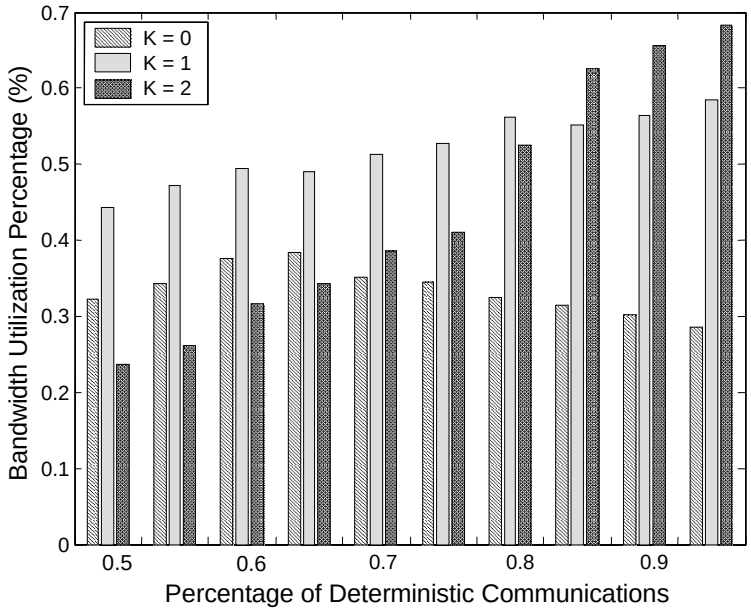


Figure 5.6: Combining preload of communication patterns with dynamic scheduling. A multiplexing degree of three was used, with k slots preloaded. k is varied from 0 to 2.

For 85% or greater determinism, the 2-preload/1-dynamic scheme performed over 10% better than the 1-preload/2-dynamic. This supports the notion of predictive communications as a hit-rate of 85% or better shows dramatic improvement in efficiency.

In this chapter, we presented the design of a real-time greedy scheduler that supports dynamic circuit establishment, compiler directed communication. The design relies on an adaptive time division multiplexing scheme. The design also allows different prediction mechanisms for pre-loading connections into the working set before they are actually needed.

6.0 OPTIMIZING SCHEDULER FOR CROSSBAR NETWORKS

Reasonable performance results can be achieved using greedy heuristics for crossbar scheduling. However, for optimized scheduling, more complex algorithms are needed. One provably optimal solution is to implement the maximum matching algorithm for bipartite graphs. The maximum matching algorithm requires $O(N^3)$ time for an $N \times N$ communication system, which limits its application to network scheduling. We reformulate this algorithm in terms of Boolean operations, rather than the original set operations and introduce three Maximum Matching Processors and show how we can trade processor complexity for performance. Specifically, we examine a Pure Logic Processor, a Matrix Processor and a Vector Processor to show how these architectures reduce the time complexity down to $O(1)$, $O(K)$, and $O(KN)$, respectively, where K is the number of optimization steps. While an optimal scheduling algorithm requires $K = 2N - 1$ steps, our simulation results show that the scheduler establishes connections 99% close to the optimal schedule when $K = 9$. We examine the hardware complexity and the time complexity of these architectures for crossbar sizes of up to $N = 1024$. Using FPGA synthesis results, we show that a greedy schedule for various sized crossbars, ranging from 8×8 to 512×512, can be optimized in less than 90 ns per optimization step.

Section 6.1 introduces prior work and research motivation. Related work on maximum matching algorithms is introduced in Section 6.2. Section 6.3 introduces some general terms and theorems. Section 6.4 describes the hardware algorithms and architectures that solve the maximum matching problem. In particular, the tradeoff of hardware timing and area cost are considered. System simulation is described with the results in Section 6.6. Finally, we discuss conclusions and future research directions in Section 6.7.

6.1 INTRODUCTION

Maximum matching in bipartite graphs is a widely studied problem [56, 57, 58, 59]. Maximum matching provides the optimal number of links between a set of sources and a set of destinations based on the requirements of system. A crossbar interconnect can be represented as a bipartite graph where each vertex corresponds to a communication node and each edge represents a link between the two nodes. Because bipartite maximum matching is optimal, the maximum network utilization is achieved. However, due to its $O(N^3)$ running time, the scalability of switching using this technique is not scalable.

Developments of electrical, analog, and optical network technologies continue to push the limits of network switches. It is now possible to place large crossbars within single chips and even larger crossbars within communication racks, as seen in the IBM BlueGene/L [10] and NEC Earth Simulator [60]. Scheduling these switches to achieve maximum throughput in the system is increasingly challenging. Currently, many of these systems use a greedy scheduling approach that results in a fast schedule, but does not achieve optimal results. If a maximum matching approach could be used, it would improve network utilization and reduce communication time.

This chapter describes three maximum matching processors. By utilizing fundamental properties of hardware logic it is possible to detect potential paths for optimization in parallel and trace back the result. As a result of implementing a Pure Logic Processor, a Matrix Processor and a Vector Processor, the algorithm run-time complexity can be reduced from $O(KN^2)$ to $O(1)$, $O(K)$ and $O(KN)$, respectively, where K is the number of optimization steps for an $N \times N$ crossbar. Additionally, by using a greedy scheduler to make initial configurations as discussed in Chapter 5, the bipartite algorithm improves its results and is able to grant more requests. By examining our simulation results, we find that near-optimal connections of approximately 99% close to the optimal results can be established in no more than nine optimization steps ($K = 9$).

6.2 PRIOR WORK

Several researchers have applied bipartite maximum matching to solve network switch configuration [61, 62]. For example, the maximum matching algorithm allows 100% throughput under uniform traffic for packet switching networks [63]. Weller and Hajek [61] applied a batch scheduling extension to maximum matching to solve the fairness problem for a crossbar network called critical maximum matching. Maximum matching can also be used to determine and utilize the optimal multiplexing degree for time-division multiplexing (TDM) networks [62]. While maximum matching provides extremely attractive network characteristics, the algorithmic complexity makes it impractical for large switches.

The sequential maximum algorithm was first proposed and proved optimal by Edmonds [64]. It iteratively searches for augmenting paths and increases the cardinality of the matching set. This has been shown to require $O(N^3)$ time with a maximum augmenting path length of $2N - 1$.

Subsequently, most sequential algorithms have been derived from the same concept [65]. Additionally, several parallel algorithms have been proposed to solve this problem. Mulmuley et. al. proposed a parallel version of the algorithm based on the pugilistic lemma requiring $O(N^{3.5})$ processors in parallel [59]. Hanckowiak et. al. implemented a parallel algorithm with computation complexity of $O(\lg^6 N)$ for approximating the maximum matching with the matching degree at least 2/3 of the maximum matching degree [66, 67]. However, to make the algorithm pratical for a scheduler within a switch, the algorithm must execute quickly *and* be able to fit within a single silicon device.

Prior works in parallelizing the maximum matching algorithm utilize multiple processors that execute sequential instructions. In contrast, the underlying algorithm is based on sets, which can be represented as binary bits. Thus, the algorithm can be implemented as a set of operates acting in parallel on these bits. Our approach utilizes parallelism amiable in a single device such as custom hardware.

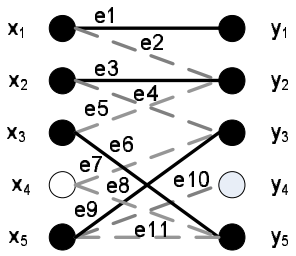


Figure 6.1: A bipartite graph representing a crossbar schedule.

6.3 BACKGROUND

In this section, we introduce general terminology and fundamental theorems related to the optimal maximum matching algorithm that will be used in its description. More detailed definitions and proofs are found in [64]. We utilize these foundational principles to create an optimizing scheduler.

Definition 1. *Bipartite graph*

A undirected graph, G, consisting of vertices and edges, $G = (V, E)$, is a Bipartite Graph if G can be separated into two sets V_1 and V_2, such that $V_1 \cap V_2 = \theta$ and $V_1 \cup V_2 = V$ and for any edge $e = (v_i, v_j) \in E$, then $v_i \in V_1, v_j \in V_2$.

Figure 6.1 gives a example of bipartite graph. $V_1 = \{x_1, x_2, x_3, x_4, x_5\}$, $V_2 = \{y_1, y_2, y_3, y_4, y_5\}$, $E = \{e_1, e_2, e_3, \ldots, e_{11}\}$.

To configure a crossbar, we need to schedule communications between pairs of nodes such that each node is only involved in sending a single message at any given time. In this case, the edges of the graph represent a communication connection between a sender and receiver. In this graph notation, each node is represented as an (x, y) pair that represents its sending and receiving capability.

Definition 2. *Maximum matching*

In bipartite graph $G = (V, E)$, for an edges set M, $M \subset E$, and there does not exist an M' such that $|M'| > |M|$ and $M' \subset E$, then M represents a maximum matching.

An objective of crossbar scheduling is to maximally utilize the routing resources available in the crossbar. We examine and optimize the scheduling of the crossbar at a particular instance in time, based on a set of requests in the system. These requests are represented as edges in the bipartite graph. Consequently, the largest possible set of edges are scheduled at each time instance.

Definition 3. *Saturated and unsaturated edges*

Given a particular matching $M \subset E$, all edges, $e \in M$ are considered to be *saturated*. All edges $e' \in E$ but $e' \notin M$ are considered to be *unsaturated*.

For crossbar scheduling, we start with a configuration and apply the maximum matching algorithm to improve the bandwidth utilization of the crossbar. In this context, an established configuration between nodes X and Y is said to be saturated. A communication request that is not granted is said to be unsaturated.

Definition 4. *Saturated and Unsaturated Vertices*

Given a particular matching $M \subset E$, all vertices that are part of a saturated edge are considered to be saturated. Conversely, all vertices that are not part of a saturated edge are considered to be unsaturated.

Definition 5. *Augmenting Path*

If a path P in G is constructed by alternating unsaturated edges and saturated edges, and both ends of the path are unsaturated, then path P is called *augmenting path*.

Theorem 1. *M is a maximum matching of G, iff G does not contain an augmenting path within M.*

Corollary 1. *If M is not a maximum matching of G, then there exists a augmenting path within M.*

Corollary 2. *Given a matching M of G and an augmenting path P, within M, then there exists a matching M' that does not contain P whose cardinality is larger than M.*

Theorem 2. *Given a matching M of G and an augmenting path P, a new matching M' can be created from M such that $|M'| > |M|$ by removing all saturated edges in P from M and by adding all unsaturated edges in P to G.*

For example, Figure 6.1 shows a matching set using the darkened edges. The augmenting path in this graph contains edges $\{e_7, e_9, e_{10}\}$. By removing the saturated edge e_9, the unsaturated edges e_7 and e_{10} can be added; thereby increasing the number of edges in the matching set.

In our hardware implementation, augmenting paths are searched in parallel using specialized processing. All computation is based on pure hardware logic, matrix or vector operations. However, asymptotic notation breaks down in this case as our architecture is comprised of $O(N^2)$ gates within a single "processor" rather than $O(N^2)$ processors. In essence, our technique utilizes a custom architecture to exploit the transistor density growth in the past decades and achieves superior performance while fitting within a single device.

One of the benefits of the maximum matching algorithm is that it continuously improves an existing matching until optimality is reached. By first utilizing a greedy strategy we establish a baseline matching. Then, by finding augmenting paths, we utilize the maximum matching algorithm to further improve the resulting solution either for a fixed amount of time or until optimality is reached.

6.4 SPECIALIZED PROCESSORS FOR OPTIMAL SCHEDULING

In this section, a parallel maximum matching algorithm is presented that can be realized with the inherent performance and density of hardware logic. All possible augmenting paths are detected in parallel. Then one path is traced back for the exact optimization. Using the algorithm, we propose three hardware designs: (1) Pure Logic Processor, (2) Matrix Processor, and (3) Vector Processor. The hardware timing and area analysis shows that based on 90nm FPGA technology, the Pure Logic Processor is feasible for small systems (16 or fewer nodes), the Matrix Processor is appropriate for large systems (32 – 128 nodes), and the Vector Processor can handle very large systems (128 – 1024 nodes).

6.4.1 Maximum matching algorithm

The main idea for hardware implementation is to unfold the bipartite graph, as shown in Figure 6.2. The graph starts with an unsaturated path and then adds multiple pairs of saturated followed by unsaturated paths.

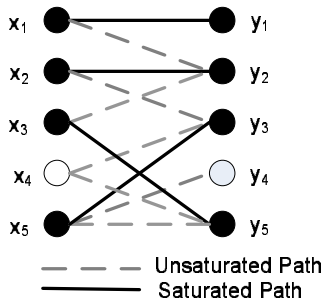
The source nodes on the left, labeled x_1 through x_5 and destination nodes on the right, labeled y_1 through y_5. Pending communications are shown as connections between X and Y nodes. Darkened connections represent an existing schedule and darkened nodes represent nodes that are involved in a communication. These scheduled connections and nodes are labeled as *saturated* while the dashed connections and nodes are labeled as *unsaturated*.

The original maximum matching algorithm uses sets to describe the algorithm and can be found in [12]. Conceptually, the algorithm removes a K saturated links (i.e., scheduled) and adds $K + 1$ links. In Figure 6.1, it can be seen that by removing $\{x_5, y_3\}$, $\{x_4, y_3\}$ and $\{x_5, y_4\}$ can be added.

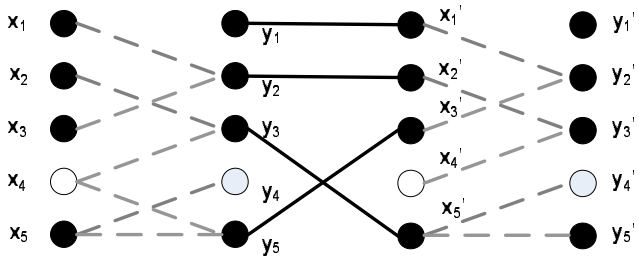
The algorithm for finding this optimization involves following an alternating sequence of unsaturated and saturated links between X and Y. The starting and ending nodes must be unsaturated. In the example in Figure 6.1, the path is $x_4 \rightarrow y_3 \rightarrow x_5 \rightarrow y_4$. The length of the path does not matter but a vertex can only be included once.

Assume that there exists an augmenting path $\{x_4, y_3\}$, $\{y_3, x_5\}$, $\{x_5, y_4\}$. We set the augmenting path as valid so that signal can pass through. If we inject a high-level signal (e.g. logic value '1') on x_4 on the left side, we should be able to detect the same high-level signal propagated through to the right side, y_4. For small sized bipartite graph, an augmenting path can be found using combinational logic in a single clock cycle.

In our representation of this algorithm, we split the graph into two graphs, labeled saturated and unsaturated. The unsaturated graph contains only unsaturated paths and the saturated graph contains only saturated paths. For the saturated path graph, we place the Y nodes on the left and the X nodes on the right. We then create a new graph by placing sequences of unsaturated then saturated graphs next to each other with overlapping vertices. Finally, an unsaturated graph is added to the far right of the graph. In essence, we flip the original graph over the Y axis an even number of times and only keep the unsaturated paths



(a) Original bipartite braph with matching M



(b) Unfolded bipartite graph with matching M

Figure 6.2: Original and unfolded bipartite graphs.

in the odd stages and only keep the saturated paths in the even stages. The optimization problem is reduced to finding a path from left to right through the graph starting and ending with an unsaturated vertex. Figure 6.2 shows the new unfolded graph with three stages. We use x'_i and y'_i to represent the unfolded vertex of x_i and y_i respectively, where $0 \leq i < N$. From this new graph, two legitimate paths existing through stages 1 and 2, namely $x_4 \rightarrow y_3 \rightarrow x'_5$ and $x_4 \rightarrow y_5 \rightarrow x'_3$, but only one of the paths traverses stage three to end at an unsaturated node, namely $x_4 \rightarrow y_3 \rightarrow x'_5 \rightarrow y'_4$. This path is equivalent to the path, $x_4 \rightarrow y_3 \rightarrow x_5 \rightarrow y_4$, as shown in the Figure 6.1.

The optimization to the schedule removes the saturated links that exist in the final path and adds the unsaturated links. In this algorithm, there is always one more unsaturated link than saturated and thus, the result is always a schedule that contains one additional link.

To determine if *any* optimizations can be found from a particular unfolded graph, we can conceptually treat the graph as if it was physically wired system and attach a voltage to the unsaturated nodes at the far left of the graph and light bulbs to the unsaturated nodes at the far right side of the graph. If any optimization path exists, then one of the light bulbs will illuminate. Thus, it can be seen that a single left-to-right pass through the graph is sufficient to determine if there are optimizations available.

There are three parts to the hardware algorithm, namely: (1) Detection of Augmenting Paths, (2) Isolation of a Single Augmenting Path and (3) Matching Set Update. Path detection is the most complex as it requires all possible paths to be checked. This is performed in parallel to discover all paths. The second pass backtracks through the graph and isolates a single path. The third step is to update the schedule (i.e., the Matching Set).

The variables used are defined as follows

- The proposed parallel maximum matching algorithm takes an iterative approach and searches for augmenting paths of increasing length. Let K be the number of steps in the original algorithm and is the maximum length of the augmenting path.
- N is the system size of an $N \times N$ crossbar.
- $F[k]$ is a matrix, representing the forward path of step k.
- $B[k]$ is a matrix representing the backtrack path of step k.
- $UnsaturatedX[i, j]$ is the mapping of unsaturated vertex i in X set to vertex j in Y set;

- $SaturatedY[i, j]$ is the mapping of saturated vertex i in Y set to vertex j in X set.
- $XMask[k]$ and $YMask[k]$ are bit-vectors of size N in step k.

Figure 6.3 shows the procedure of detecting augmenting paths. We use a bit-vector $XMask[k]$ to represent the status of nodes at step k. If x_i is unsaturated, the i^{th} bit in XMask[k] is set as '1'; otherwise, it is set as '0'. In this example, only x_4 is unsaturated after greedy scheduling, therefore $XMask[1]$ is {0 0 0 0 1 0}. We use $XMask[1]$ to filter $UnsaturatedX$. If the i^{th} bit of XMask is '1', the i^{th} row of Unsaturated X is enabled. In Figure 6.3, only the 4^{th} bit in XMask is '1', therefore only the 4^{th} row of UnsaturatedX is enabled. This is equavelent to pairwise vector matrix multiplication. The resulting matrix is $F[1]$. $Xmask[2]$ is constructed by columnwise OR reduce. Similarly, after filtering SaturatedY using Xmask[2], we get $F[2]$. By applying OR on each column of $F[2]$, we get $XMask[3]$, which after a column-wise OR reduce becomes $F[3]$. We define a vertex as active if it is traversed during discovery of augmenting paths. y_3 and y_5 are active after first step. $x3'$ and $x5'$ are active in the second step and y'_2, y'_4, and y'_5 in the final step. The algorithm terminates as y'_4 is an unsaturated vertex. The next step in the algorithm is to isolate the augmenting path and update the matrices. The optimization then begins again with the next matrices.

Once one or more augmenting paths have been found, exactly one of these paths must be identified. This requires backtracking through the F matrices and selecting a single path to isolate along the way. It is possible to isolate multiple paths but for simplicity, we isolate only a single path. Figure 6.4 shows the backtracking to isolate a single augmenting path. We use YMask to represent the status of nodes. We start from the most right side for backtracking. Because y'_4 is unsaturated, the 4^{th} bit of YMask[3] is '1'. If there are multiple bits in YMask are '1'. We only pick up one bit. In our design, we pick the lowest possible bit. We use that bit to enable the column of F[3]. B[3] is the result after perform filtering. After applying 'or' logic on each row of B3, we get YMask[2]. YMask[2] are used to filter F[2] and generate B[2]. Following the same approach, B[1] is generated. In graphical point of view, x'_5 is active after first step. y_3 is active in the second step and x_4 in the final step. The augmenting path is isolated, which is $y'_4 \rightarrow x'_5 \rightarrow y_3 \rightarrow x_4$. It is equivalent to the path $y_4 \rightarrow x_5 \rightarrow y_3 \rightarrow x_4$ in the original graph.

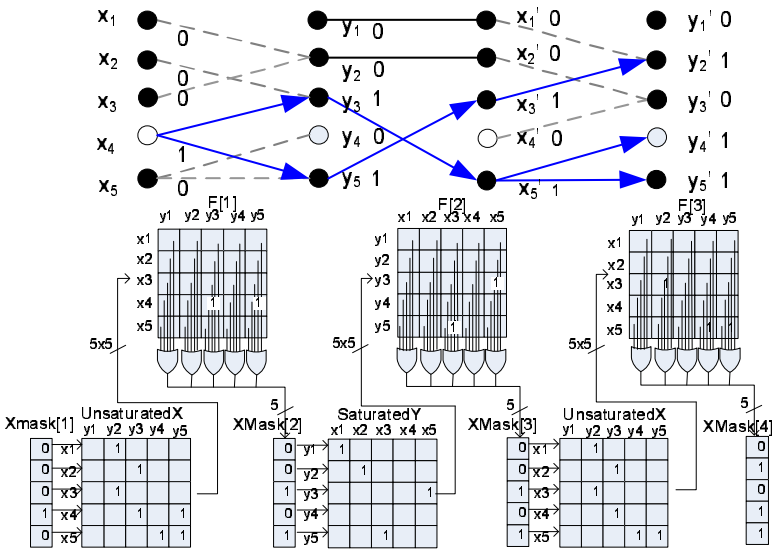


Figure 6.3: Parallel tracing of potential augmenting paths as described in detection of augmenting paths algorithm

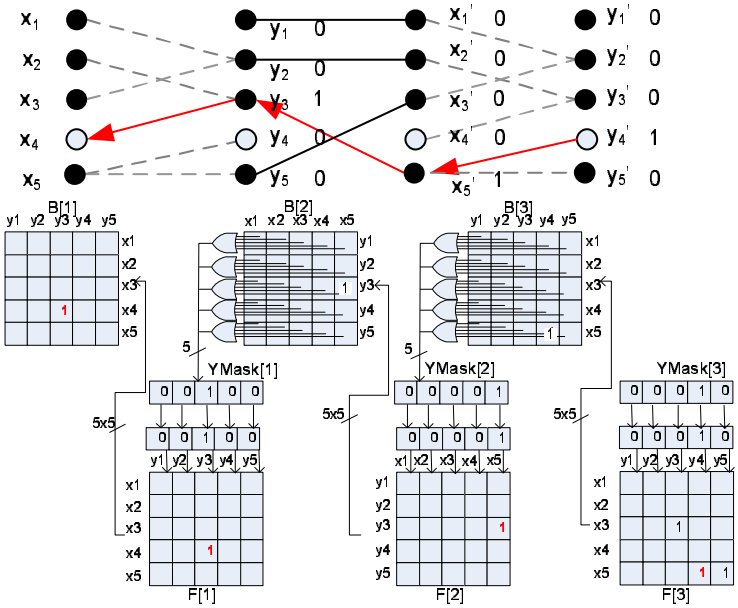


Figure 6.4: Isolation of a single path within the augmenting paths.

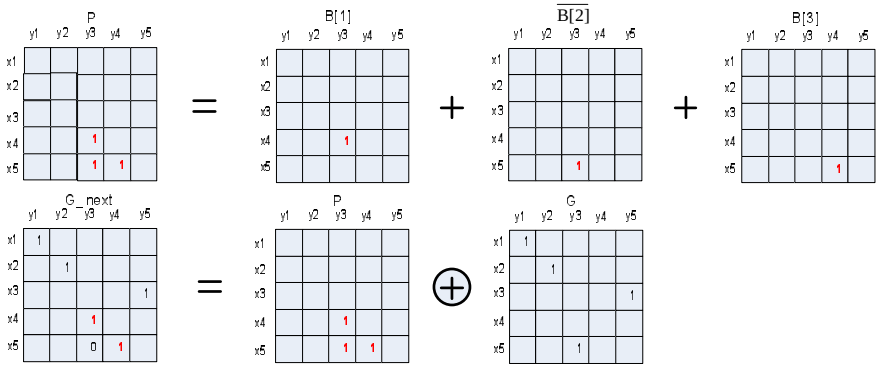


Figure 6.5: Matching set update.

The final step after isolating a single augmenting path is to update the matrices. The updating of the connection matrices is illustrated in Figure 6.5. All unsaturated edges in the augmenting path are added to the matching set and all saturated edges in the augmenting path are removed from the matching set. Thus, all odd steps in the backtracking algorithms B[1, 3, 5, ..] are added while the even steps are removed. We first generate and store the augmenting path, which is a matrix P. P is generated by applying 'and' logic on B1,transpose of B2, and B3. Then the optimized matching is achieved by applying XOR logic on P and the original matching matrix G.

Our proposed algorithm is summarized below. The pseudo code is described in a general form that can be implemented by pure hardware logic, matrix, or vector operations. We describe the tradeoffs and performance of each of these three options in the following section.

The detection of Augmenting Paths Algorithm shown in Figure 6.6 is traced forward through the unfolded graph in parallel. Line 2 creates a vector that contains unsaturated vertices. The UnsaturatedX matrix is a connection matrix and by using the XMask, we remove all paths that do not originate from an unsaturated vertex. Line 4 determines which vertices in second column are active, that is, those vertices that are part of a path. This is performed by a OR'ing the columns in the matrix. These three lines represent a single step in the *unfolded* graph.

```
1.  // Start with an unsaturated path
2.  XMask[1] = UnsaturatedXVertices
3.  i = 1;
4.  while Not PathFound and i ≤ K {
5.      // Follow a Unsaturated path
6.      F[i] = XMask[i]*UnsaturatedX
7.      XMask[i+1] = ColumnwiseReduceOR(F[i])
8.      // See if any augmenting path is found
9.      PathFound = ReduceOr (XMask[i+1] * UnsaturatedYVertices )
10.     // Follow an Saturated path
11.     F[i+1] = Xmask[i+1]*SaturatedY
12.     XMask[i+2] = ColumnwiseReduceOR(F[i+1])
13.     i=i+2
14. }
```

Figure 6.6: Detection of augmenting paths algorithm.

```
15. S = Final step number
16. // Select one of the active unsaturated vertices
17. YMask[s] = SelectOne(XMask[S])
18. for (i = S; i > 1; i−−) {
19.     B[i]=YMask[i]*F[i]
20.     YMask[i]=SelectOne( RowwiseReduceOR(B[i]) )
21. }
```

Figure 6.7: Isolation of a single augmenting path.

The loop beginning on line 5 represents two steps in the unfolded graph, the first being through saturated edges and the second being through unsaturated edges. This loop can iterate multiple times as the graph can be unfolded numerous times.

It can be seen that the operations required for the maximum matching algorithm are vector are matrix operations. We further observe that the data types required are all Boolean and are relatively simple when compared to a traditional processor. However, traditional processors do not perform matrix operations well, even when they are Boolean. Thus, we will next explore three alternative architectures for acceleration of these algorithms.

6.5 HARDWARE TIMING AND AREA COST

The maximum matching can be processed in one clock cycle using pure combinational logic or in multiple cycles by using either matrix or vector operations. We first examine the asymptotic tradeoffs and then describe each of three architectures.

We observe that for small values of N, a purely hardware version of the maximum matching can be performed in a single cycle. This is the Pure Logic Processor. The Matrix

Processor performs each of the steps in the maximum matching and path isolation algorithms in a single cycle. This requires a register file for storing the intermediary results. The Vector Processor is focused on large values of N where it is not possible to have an $O(N^2)$ register file. This architecture uses traditional memories for storage.

According to the hardware structure, the hardware cost can be computed approximately as shown in Table 6.1, where N represents the number of nodes and K represents the number of optimization steps.

Forward path logic and backtrack logic consumes most of the logic area and running time. The time delay is mainly caused by the forward path logic and backtrack logic also. However, the delay for forward path and backtrack path is different. The forward path propagates high-level signal forward, vector computations such as 'or' and 'and' are involved. The backtracking path is responsible to select one augmenting path from all candidates. A priority selector is used, which add extra delay to the backtrack path. In such condition, we can detect possible improvement by only applying forward path logic. Then, if potential improvement is detected, backtrack path logic will keep working, otherwise, we just stop at the forward path logic and save the rest of the computation time.

6.5.1 Pure logic processor

In the Pure Logic Processor, the maximum matching operation is performed by pure combinational logic, as shown in Figure 6.8. A sequence of forward logic blocks is responsible for detection of the augmenting paths while a sequence of backward logic blocks is responsible for isolation of a single augmenting path. Both the forward and backward logic blocks are combined together to form a single block of hardware. This consumes a large amount of logic and routing resources. However, once signals propagate the circuit, the optimization result is achieved. Therefore, for relatively small number of optimization steps and graph sizes, this is effective.

Table 6.1: Complexity analysis of three maximum matching architectures. N presents the number of nodes in the system and K represents the number of optimization steps performed.

	Architecture		
	Pure logic	Matrix processor	Vector processor
Wire compexlity	$O(KN^2)$	$O(N^2)$	$O(N)$
Logic complexity	$O(KN^2)$	$O(N^2)$	$O(K)$
Memory complexity	0	$O(KN^2)$	$O(KN^2)$
Latency in cycles	$O(1)$	$O(K)$	$O(KN)$
Multiple iterations	Disable	Enable	Enable

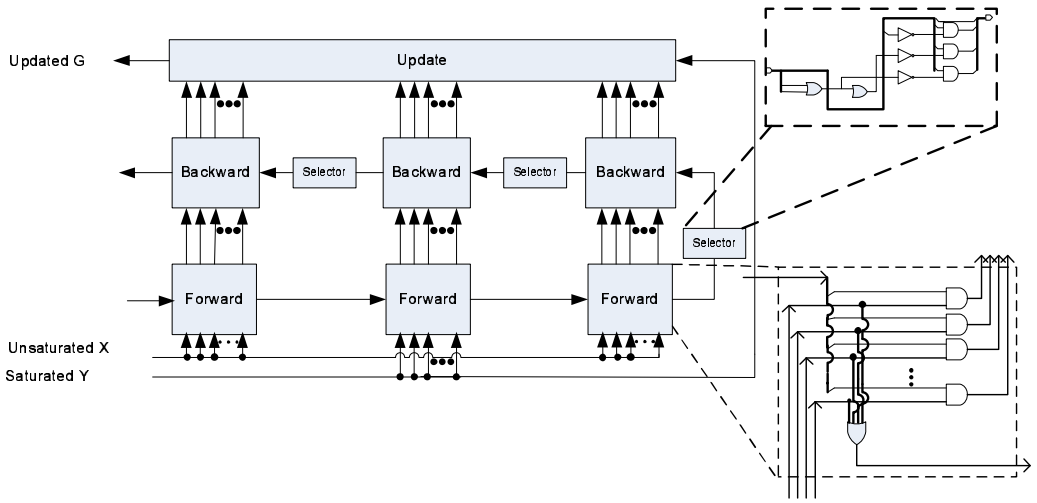


Figure 6.8: Pure Logic Processor to implement the maximum matching algorithm.

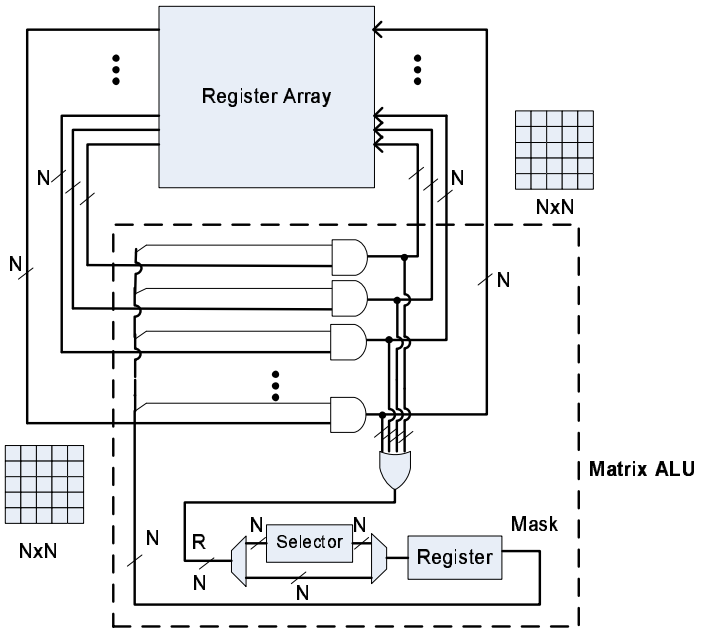


Figure 6.9: Matrix Processor for the maximum matching algorithm.

6.5.2 Matrix processor

The Matrix Processor breaks the algorithms down into matrix operations. Each of the steps in the algorithms is performed by a Matrix ALU in a single cycle. Matrices are stored in a register array. At each cycle, one matrix is read from the register array. The mask bits enable outputs of each row in the original matrix by N 'or' gates. Each of the 'or' gate is $N + 1$-bit wide. ReduceOR is performed on the outputs using one NxN-bit 'or' gate and generates an N-bit signal R, as shown in Figure 6.9. R is XMask vector when the Matrix ALU operates for detection augmenting paths. When performing isolation step, R is filtered by a selector to generate YMask vector.

With the Matrix ALU, the original algorithm is implemented as matrix operations. Let us use MatrixALU( ) to represent the function performed by the Matrix ALU. For example, the code from line 4 to line 14 in Figure 3 is written as shown in Figure 10 by using MatrixALU().

```
4.  while Not PathFound and i < K {
5.      // Follow a Unsaturated path
6.      (F[i], XMask[i+1]) = MatrixALU(Xmask[i], UnsaturatedX)
7.      // See if any augmenting path is found
8.      PathFound = ReduceOr (XMask[i+1] * UnsaturatedYVertices )
9.      // Follow an Saturated path
10.     (F[i+1], XMask[i+2]) = MatrixOp(Xmask[i+1], SaturatedY)
11.     i = i + 2
12. }
```

Figure 6.10: Detection of augmenting paths algorithm using matrix operations.

This is more scalable and only requires $O(N^2)$ hardware. For relatively moderately sized N, this is still reasonable.

6.5.3 Vector processor

The Vector Processor implements N-wide memory reads and vector operations in a single cycle. This architecture utilizes traditional memory as current ASIC devices contain over one thousand I/O pins and commercial memories can be width expanded by sharing the address lines among multiple banks of SRAM. Figure 6.11 gives a block diagram of the Vector Processor. During each cycle, a vector of a matrix is read from the memory. The output of the vector is enabled by one bit of a Mask. The bit is retrieved from XMask and YMask, for detection augmenting paths and isolation an augmenting path respectively. One mask bit from a Mask vector is output at each cycle. The process for generating XMask and YMask is similar as that in the Matrix Processor. However, it requires N cycles to process a complete matrix and then generate a Mask vector, while it only need one cycle in the Matrix Processor.

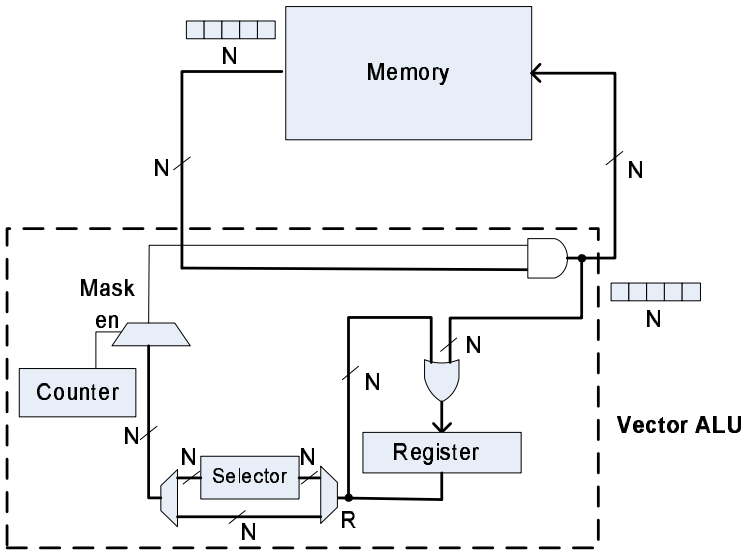


Figure 6.11: Vector Processor for the maximum matching algorithm.

We use VectorOp( ) to represent the function performed by the Vector ALU. The maximum matching algorithm from line 4 to line14 in Figure 6.6 is written as shown in Figure 6.12 by using VectorALU( ).

It requires N cycles to perform one optimization step. However, our performance evaluation results in Section 6.6 show that this architecture scales the best for large sizes of N.

6.6 PERFORMANCE EVALUATION

We built a systems consisting of 16, 32, 64 and 128 nodes and estimated for 512 and 1024 nodes. The hardware cost evaluated based on Altera Stratix ESP1S25F1020C FPGA chip. The hardware synthesis result may different if different fabric methods are applied. The maximum number of logic elements and memory bits in ESP1S25F1020C FPGA chip are shown by dashed lines in Figure 6.14 and Figure 6.15 respectively.

```
4.  while Not PathFound and i < K {
5.      // Follow a Unsaturated path
6.      For(j = 0; j < N; j++) {
7.          (F[i][j], XMask[i+1][j]) = VectorALU(Xmask[i][j], UnsaturatedX[j])
8.      }
9.      // See if any augmenting path is found
10.     PathFound = ReduceOr (XMask[i+1] * UnsaturatedYVertices )
11.     // Follow an Saturated path
12.     For(j = 0; j < N; j++) {
13.         (F[i+1][j], XMask[i+2][j]) = VectorOp(Xmask[i+1][j], SaturatedY[j])
14.     }
15.     i=i+2
16. }
```

Figure 6.12: Detection of augmenting paths algorithm using vector operations.

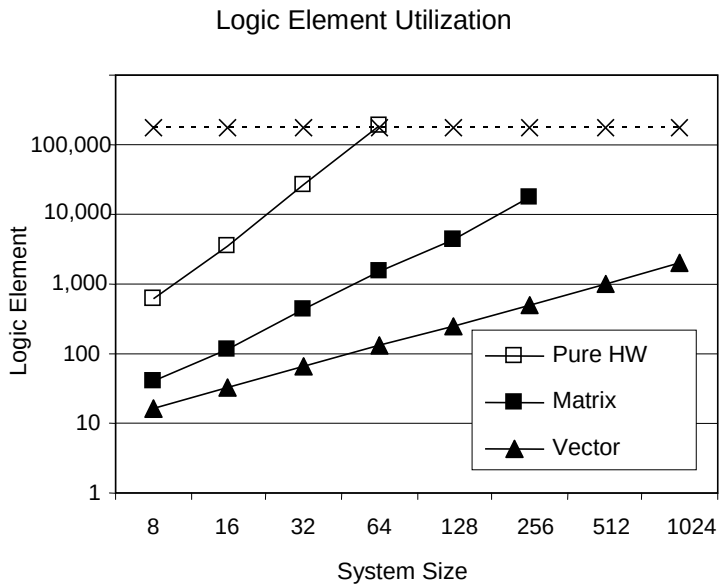


Figure 6.13: Performance per optimization step. The Pure Hardware performance is based on estimations. The Vector and Matrix performance numbers are based on actual hardware synthesis results ranging from 8-128 and estimated for 512 and 1024.

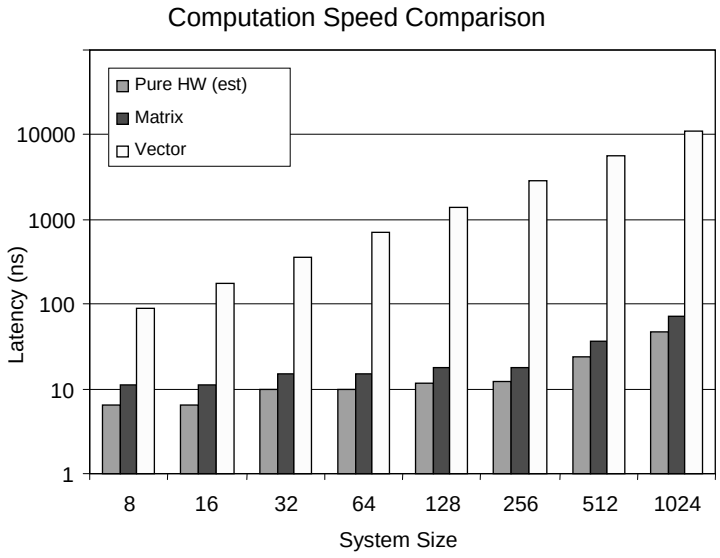


Figure 6.14: System area cost. The Pure Hardware performance is based on estimations of a single optimization step. The Vector and Matrix performance numbers are based on actual hardware synthesis results ranging from 8-128 and estimated for 512 and 1024.

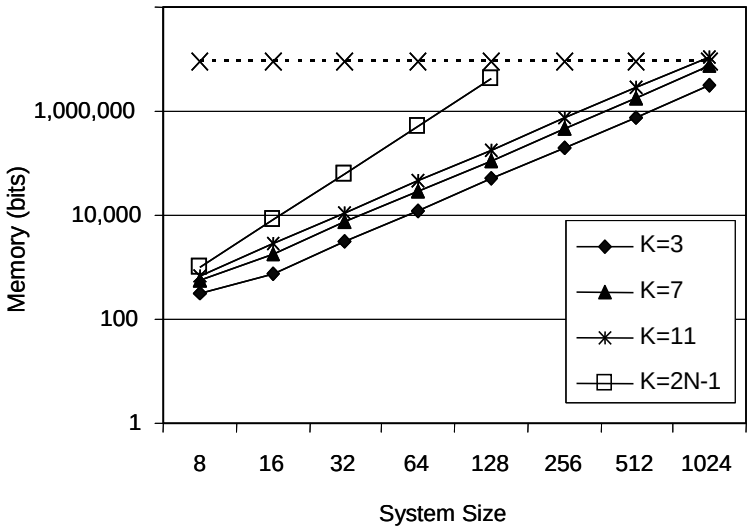


Figure 6.15: Estimated memory utilization for various step sizes, K, (with K=2N-1 steps being provably optimal.)

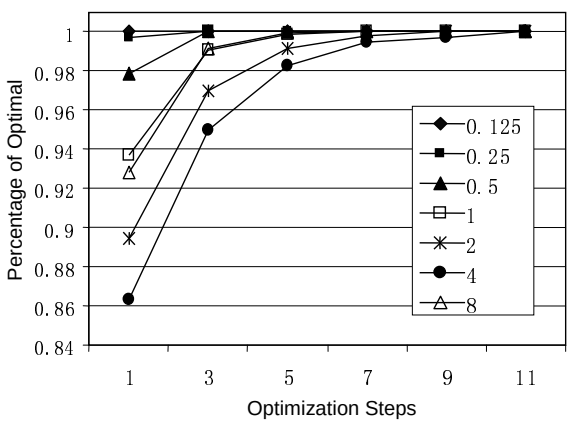
If only considering processing delay, it can be seen that pure combinational logic and register array architecture have similar performance and both outperform memory architecture, as shown in Figure 6.13. Nevertheless, the area cost can not be neglected. Figure 6.14 indicates that register array based architecture consumes less logic elements compared with pure combinational logic architecture. Memory based architecture requires the least logic elements, which is fit for large-scaled design. We conclude that combinational logic works well for system containing less than 64 nodes; for system containing 64-128 nodes, register array based architecture should be chosen; memory based architecture support system up to 1024 nodes.

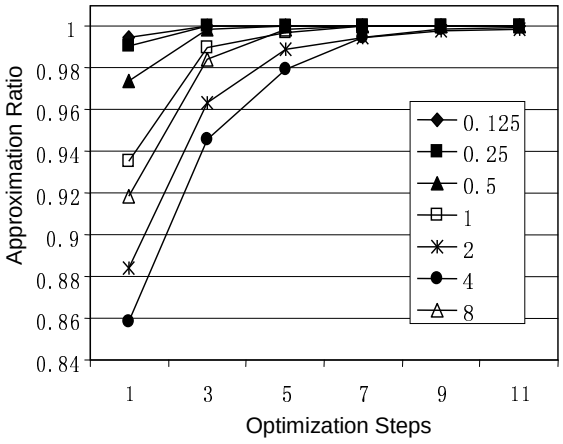
The critic process is to find the augmenting path for optimization. This process is implemented by three main hardware logic blocks as shown earlier, which is forward logic, backward logic and update logic. All of them are implemented by pure combinational logic. The data is stored and transferred through wire. The architecture guarantees the logic is passed through within one cycle. However, when the hardware design is complete, the number of iterations, in another word, the number of optimization steps is fixed. If system

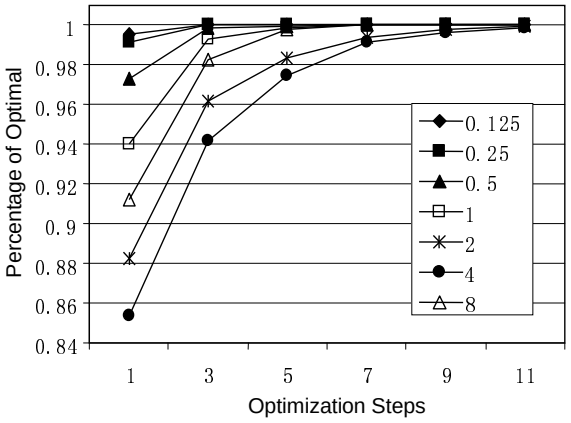
requires more optimization steps, more hardware logic has to be added before processing. It's better to make the design more flexible, so that the design is able to support multiple iterations.

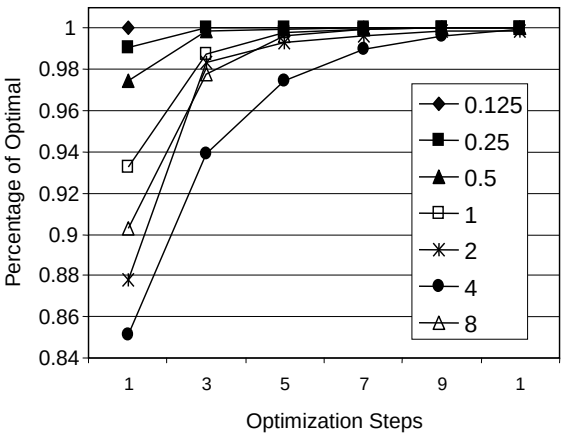
Our solution is to put data storage unit into the design, like register array and memory, shown in Figure 6.14. The difference between register array and memory is that register array output complete matrix at one cycle; however, memory is able to output a bit vector each cycle. The register array based design supports matrix operation while the memory based design support vector operation only. We call them matrix operation architecture and vector operation architecture separately. Since one matrix operation invokes N times vector operation, where N is the size of the matrix. Therefore, the computation time derived by matrix operation architecture and vector operation architecture is quite different.

We performed system simulations for communication matrices with/without complete matching based on system containing 16, 32, 64, and 128 nodes. We generated random communication matrices with various densities. For example, in a 8x8 communication system, if totally four requests are generated randomly, we mark the density as 4/(8x8), which is 0.125. The experiments are repeated 100 times, Figure 6.16 shows the matching performance corresponding to different optimization steps. Each line represents the results generated by communication matrices of a certain density. The simulation results show that for random requests, the matching degree we get is approximately 99% close to the optimal matching degree. In order to verify our hardware implementation is capable of finding complete matching, we use mixed communication requests to evaluate the system performance. The mixed communication requests are generated by combining random permutations with random requests. Our proposed algorithm is able to optimize matching successfully. The results show that for mixed requests, our matching degree is approximately 99% of the maximum matching degree after nine optimization steps, as shown in Figure 6.17.





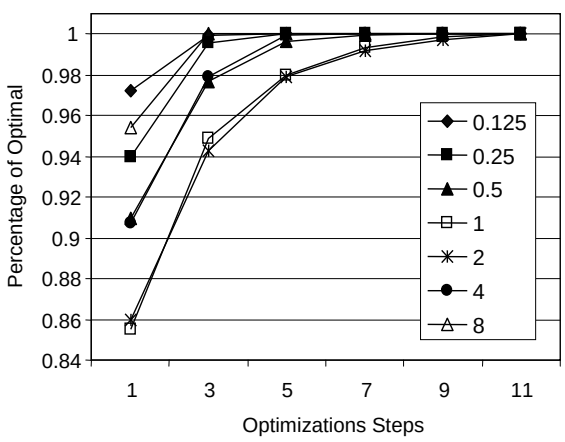


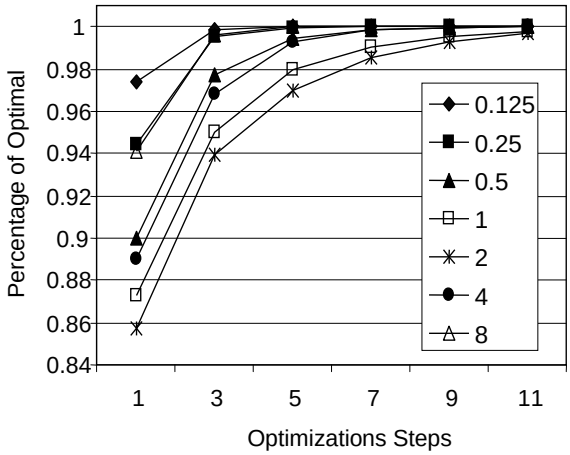


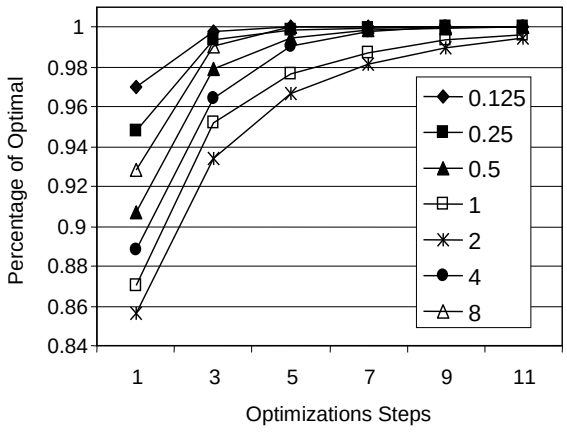
(c) Maximum matching for 64 nodes.

(d) Maximum matching for 128 nodes.

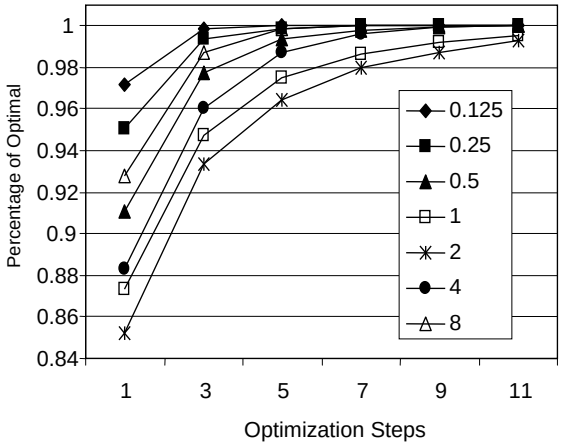
Figure 6.16: Maximum matching for random requests. The different curves represent network load where 0.125 is 12.5% loaded and 8 is 800% overloaded ($K = 1$ represents the greedy algorithm).







(c) Complete matching for 64 nodes.



(d) Complete matching for 128 nodes.

Figure 6.17: Complete matching. Requests are mixed by randomly generated request at variable loads with the oversubscribed network loads being randomly generate permutation ($K = 1$ represents the greedy algorithm).

6.7 CONCLUSION

We present a hardware design architecture which solves the maximum matching with low complexity. In our proposed hardware algorithm, we unfold the bipartite graph, so that the unsaturated path and saturated path appears alternatively. By using the inherent characteristic of hardware logic, all possible augmenting are detected in parallel. We apply the hardware algorithm into three maximum matching processors, which are Pure Logic Processor, Matrix Processor and Vector Processor.

The hardware maximum matching processor can be attached with hardware switch scheduler to provide higher network utilization without adding too much latency. With current FPGA technology, it is possible to handle maximum matching capability for up to 1024 nodes. For small to medium sized system, the processing delay for one optimize step is less than 10ns. Our current design and simulation is based on single-stage crossbar. Our future research will apply the algorithm on multiple interconnected switch networks to help make more efficient scheduling decision.

7.0 LEVEL-WISE SCHEDULING FOR FAT TREE INTERCONNECTION NETWORKS

This chapter presents efficient hardware architectures for scheduling connections on a fat-tree interconnection network for parallel computing systems. Our technique utilizes global routing information and selects upward routing paths so that most conflicts can be resolved. Thus more connections can be successfully scheduled compared with a local scheduler. As a result of applying our technique to two-level, three-level and four-level fat-tree interconnection networks of various sizes in the range of 64 to 4096 nodes, we observe that the improvement of *schedulability ratio* averages 30% compared with local scheduling. Our technique is also scalable with increased benefits for large system sizes.

Section 7.1 introduces prior work and motivation. Section 7.2 provides a background on fat trees and presents formal notation that will be used to describe the algorithms. Key observations and theorems are also given. Section 7.4 describes the Level-Wise fat-tree scheduling algorithm in detail. Section 7.5 shows the system simulation results. FPGA-based hardware that efficiently implements the Level-Wise algorithm is introduced in Section 7.6. Conclusions are offered in Section 7.7.

7.1 INTRODUCTION

Fat-tree interconnection networks have several good features, including scalability and simple topology. This network architecture was first proposed by Leiserson [68] as a hardware efficient and general purpose interconnection network. It has the structure of a tree and its links have different bandwidth at each level. The closer a link is to the root, the larger

the bandwidth. In a fat-tree network, all processing elements are located at the leaf nodes while the paths are routed through intermediate switching nodes. It has been proven to be an area-universal interconnection network [69]. Fat-tree interconnect networks can simulate any other network topology with the same silicon area with at most poly-logarithmic slowdown [69]. Whereas, other network topologies, including 2-D arrays and simple trees, will see polynomial slowdown when simulating other networks. Due to those advantages, the fat-tree topology has become a popular network architecture for massively parallel computing systems such as the Thinking Machine CM-5 [69, 70], Meiko supercomputer CS-2 [71], COMPAQ AlphaServer SC [72] and Quadrics QsNetII [73].

By convention, the scheduling approaches for fat-tree interconnection network are developed for store and forward and wormhole routing. Therefore, almost all of the scheduling algorithms are based on the local knowledge within switch nodes. Adaptive distributed scheduling is a widely used scheme for scheduling communications in fat-trees [74, 75]. In this scheme, each switch selects a routing path randomly from the available local ports. Based on this scheme, heuristic routing algorithms are developed, such as Turn Back When Possible (TBWP) algorithm proposed by Kariniemi and Nurmi [76]. They suggest connecting the top-most switches together and keep forwarding a request upward until the top-most switch if the request can not be forwarded back to its destination.

Scheduling approaches that use local information provide good scalability. However, the success of routing in a local switch node does not imply the success of routing in the entire network. We define the *schedulability ratio* to be the number of successful connections divided by the number of total requests. It can be seen that when scheduling with local routing information, the effort put into scheduling one switch node may not help improve the schedulability ratio of the entire network. schedulability ratio impacts bandwidth utilization. If the scheduling ratio is far below optimal, the bandwidth utilization is inefficient. Especially for long-lived connections, the penalty of low bandwidth utilization detrimentally impacts execution time.

To solve the problem of maximizing the schedulability ratio in fat-tree interconnection networks, we make a few key observations about the structure of a fat-tree network and develop the Level-Wise Fat Tree scheduling algorithm. Specifically, we observe that all

switch configuration options are set once a message reaches the top of the fat tree; that is to say that there is only one path between the top of a fat tree and its destination. We also observe that the upward path from the source switch to the top of the fat tree is symmetrical with the downward path from the top of the fat tree to the destination. The Level-Wise Fat Tree scheduling algorithm leverages this symmetry at each level to allocate both the upward path and downward path simultaneously. Using only local switch information, our simulation results show schedulability ratio of 45%-70%, depending on the size and depth of the fat-tree. The Level-Wise Fat Tree scheduling algorithm is able to provide schedulability ratio of 78%-95%.

7.2 BACKGROUND

A fat-tree interconnected network is denoted by FT(l, m, w), where l is the number of levels in the tree, m is the number of children in each switch node and w is the number of parents of each switch node. Fat-tree FT(l, m, w) has l levels of switches. Each level h has w^{l-1} switch nodes. Each switch is labeled by SW(h, τ), where $h = 0, 1, \ldots, l-1$ and $\tau = 0, 1, \ldots, w^{l-1} - 1$. We use a bit vector to represent τ in order to more clearly describe algorithms later. $t_{l-2}t_{l-3}\ldots t_0$ is the base-w representation of the integer τ, $0 \leq \tau < w^{l-1}$. Let $t_i = \tau$ div w^i, then $\tau = \sum_{i=0}^{l-2} t_i w^i$.

In general fat tree interconnection networks, each switch node is symmetrical, therefore m equals w. The symmetrical fat tree is detonated as FT(l, w), which is the assumption for the proof of the Level-Wise scheduling algorithm. However, the algorithm is also applicable when m and w are not equal.

7.3 FAT-TREE CONSTRUCTION

The fat-tree architecture is constructed recursively as introduced by Ohring [16]. Let FT(l, w) be a fat-tree with l levels, and each switch node has w children and w parents. FT$(l+1, w)$

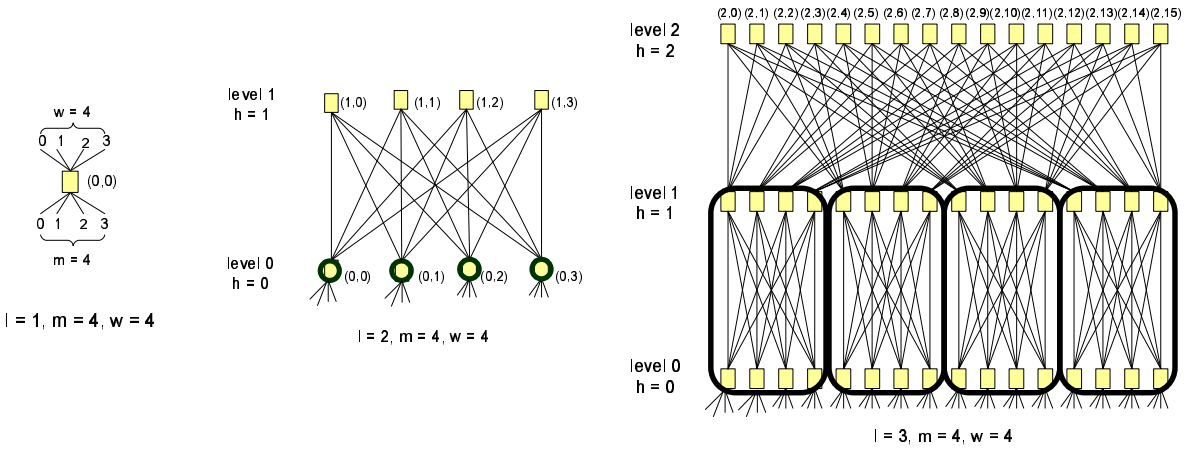


Figure 7.1: Fat-tree construction.

is built from w copies of FT(l, w) and w^l additional switch nodes. The top switch nodes in each copy of FT(l, w) are connected with the w^l additional switch nodes. The SW(l, τ) is connected with SW$(l+1, (\tau \times w) \mod w^l)$, SW$(l+1, (\tau \times w) \mod w^l + 1), \ldots,$ SW$(l+1, (\tau \times w) \mod w^l + w - 1)$.

Each switch node has w bi-directional links connected with the adjacent upper level and w bi-directional links connected with the adjacent lower level. The selection of ports decide the link between two levels, hence we use Ulink(h, τ, i) to represent upward link connected through port i in switch (h, τ) and Dlink(h, τ, i) to represent downward link connected through port i in switch (h, τ), as shown in Figure 7.2. We use a pair of vectors, Ulink$(h, \tau)[i]$ and Dlink$(h, \tau)[i]$, to represent the availability of upward and downward links, where $i = 0, 1, \ldots, w - 1$. If Ulink$(h, \tau)[i]$ equals one, upward link connected via port i of switch (h, τ) is available; otherwise, it is occupied. We use P_h to represent the upper port number selected at level h. When performing routing inside a switch node, a communication request from the source will be routed upward through one of the upward links until it reaches a switch which is a common ancestor of both the source and destination; then, the request will be routed downward through a downward link toward the destination [74, 75].

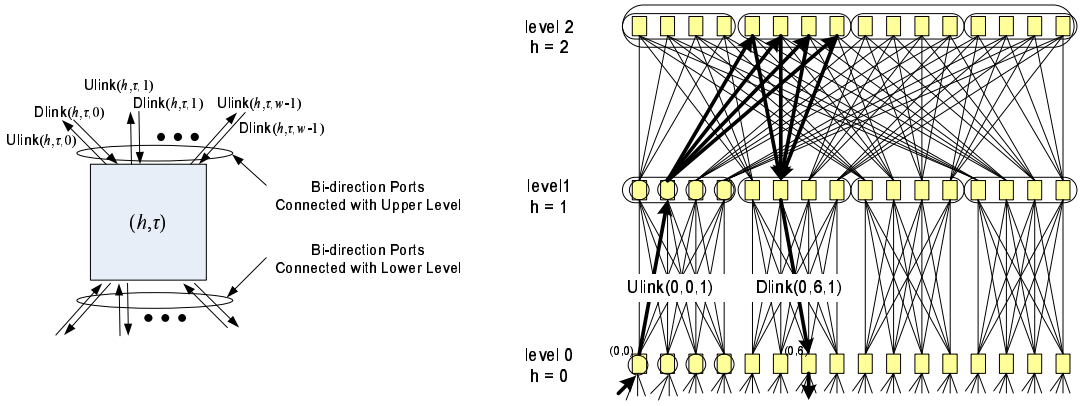


Figure 7.2: The link selection.

Theorem 3. *For switch* (h, τ_h), $\tau_h = \sum_{i=0}^{l-2} t_i w^i$, *if the ports* P_h *for level* $h(h > 0)$ *is chosen, then the switch at level* $h+1$ *that connected with switch* (h, τ_h) *is switch* $(h+1, \sum_{i=h+1}^{l-2} t_i w^i + \sum_{i=1}^{h} t_{i-1} w^i + P_h)$.

Proof:

Assume the switch at level h is $\mathrm{SW}(h, \tau_h)$, and the $\mathrm{SW}(h+1, \tau_{h+1})$ at level $h+1$ is connected with it. According to the fat tree construction rule, $\mathrm{SW}(h, \tau_h)$ is connected with $\mathrm{SW}(h+1, \tau_{h+1})$ within one $\mathrm{FT}(h+1, w)$. Therefore the value of τ_h consists of two parts, Γ_h and Δ_h. Γ_h is counted from the most left till the beginning of the $\mathrm{FT}(h+1, w)$. Δ_h is calculated as in the $\mathrm{FT}(h+1, w)$. The value of Γ_{h+1} is not affected by the selection of P_h, while the value of Δ_{h+1} is determined by P_h.

$$\tau_h = \Gamma_h + \Delta_h \tag{7.1}$$

$$\Gamma_h = \left(\tau_h \text{ div } w^{h+1}\right), \Delta_h = \tau_h \mod w^{h+1} \tag{7.2}$$

$$\tau_{h+1} = \Gamma_{h+1} + \Delta_{h+1} = \Gamma_h + \Delta_{h+1} \tag{7.3}$$

$$\Delta_{h+1} = [\Delta_h w + P_h] \mod w^{h+1} = \left[\left(\tau_h \mod w^{h+1}\right) w + P_h\right] \mod w^{h+1}. \tag{7.4}$$

Therefore

$$\begin{aligned}
\tau_{h+1} &= \left(\tau \text{ div } w^{h+1}\right) w^{h+1} + \left[\left(\tau_h \mod w^{h+1}\right) w + P_h\right] \mod w^{h+1} \\
&= \sum_{i=h+1}^{l-2} t_i w^i + \left(\sum_{i=0}^{h} t_i w^{i+1} + P_h\right) \mod w^{h+1} \\
&= \sum_{i=h+1}^{l-2} t_i w^i + \sum_{i=1}^{h} t_{i-1}^h w^i + P_h
\end{aligned} \tag{7.5}$$

Q.E.D.

The selection of the upward link determines the downward link. Specially, it will be proven in Theorem 4 that if a request is routed upward using Ulink(h, σ_h, P_h)at level h, it will reach its destination switch using Dlink(h, δ_h, P_h) at the same level. In the example shown in Figure 7.2, we show a FT$(3, 4)$ and consider a communication request from SW(0,0) to SW(0,6). The communication request can be routed through Ulink$(0, 0, 0)$, Ulink$(0, 0, 1)$, Ulink$(0, 0, 2)$, or Ulink$(0, 0, 3)$. If $P_0 = 1$, then Ulink$(0, 0, 1)$ is selected at switch (0,0) for upward routing, the request will be routed back to level 0 using the same port number, which is 1, no matter what routing path is selected above level 0. The downward link is Dlink$(0, 6, 1)$. This phenomenon is explained and proved in Theorem 4.

Theorem 4. *Given a source* SW$(0, \sigma_0)$ *and a destination* SW$(0, \delta_0)$*,assume that the common ancestor of both switches is at level* H*, where* $H < l$*. If* $P_0, P_1, \ldots, P_{H-1}$ *are the upper port numbers selected at level 0, level 1, ... and level* $H - 1$ *for upward path from* SW$(0, \sigma_0)$ *to* SW(H, σ_H)*, then the backward path from* SW(H, σ_H) *to* SW$(0, \delta_0)$ *uses the same upper port numbers* $P_0, P_1, \ldots, P_{H-1}$*, but in different switches.*

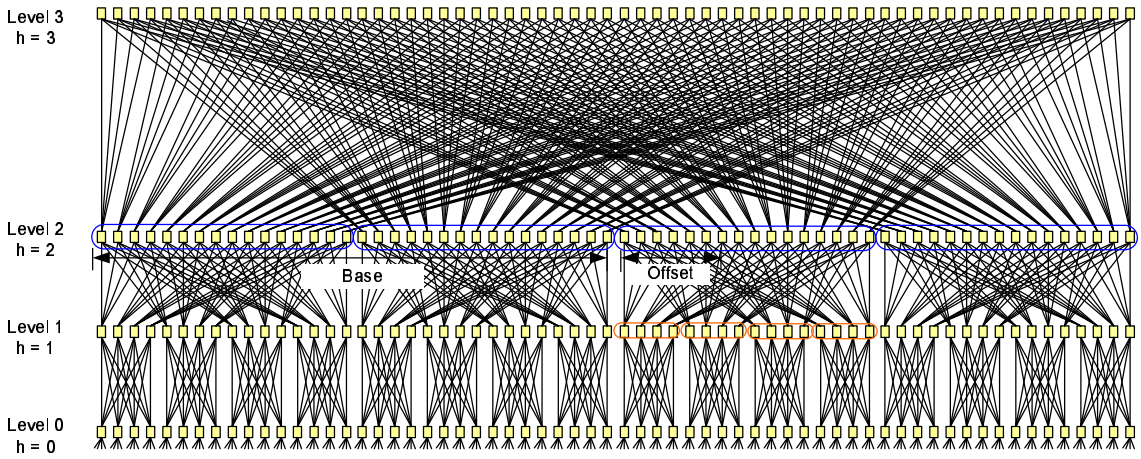


Figure 7.3: Switch node computation.

Proof:

According to theorem 1, as

$$\tau_0 = \sum_{i=0}^{l-2} t_i w^i \tag{7.6}$$

Let $h = 0$, substitute (7.6) into (7.5), we obtain

$$\tau_1 = \sum_{i=1}^{l-2} t_i w^i + P_0 \tag{7.7}$$

Let $h = 1$, substitute (7.7) into (7.5), we get

$$\begin{aligned} \tau_2 &= \sum_{i=h+1}^{l-2} t_i w^i + \left(\left(t_1 w^1 + P_0\right) w + P_h\right) \mod w^2 \\ &= \sum_{i=2}^{l-2} t_i w^i + P_0 w + P_1 \end{aligned} \tag{7.8}$$

Similarly, we get

$$\tau_{h+1} = \sum_{i=h+1}^{l-2} t_i w^i + P_0 w^h + P_1 w^{h-1} + \cdots + P_h = \sum_{i=h+1}^{l-2} t_i w^i + \sum_{i=0}^{h} P_h w^{h-i} \tag{7.9}$$

Let SW$(0, \sigma_0)$ be the source switch and SW$(0, \delta_0)$ be the destination, where $\sigma_0 = \sum_{i=0}^{l-2} s_i w^i$ and $\delta_0 = \sum_{i=0}^{l-2} d_i w^i$. As we assume that the communication ancestor of both switches is at level H, where $H < l$, switch (H, σ_H) equals switch (H, δ_h).

Since $\sigma_0 = \sum_{i=0}^{l-2} s_i w^i$, and $P_0, P_1, \ldots, P_{H-1}$ are the port numbers selected at level 0, level 1, ..., and level $H-1$ for upward path,

$$\sigma_H = \sum_{i=H}^{l-2} s_i w^i + \sum_{i=0}^{H-1} P_i w^{H-1-i} \tag{7.10}$$

Assume a message is forwarded upward from $(0, \delta_0)$ to (H, σ_H), where $\delta_0 = \sum_{i=0}^{l-2} d_i w^i$. We also assume $P'_0, P'_1, \ldots, P'_{H-1}$ are the port numbers selected at level 0, level 1, ..., and level $H-1$ for upward path.

$$\delta_H = \sum_{i=H}^{l-2} d_i w^i + \sum_{i=0}^{H-1} P'_i w^{H-1-i}. \tag{7.11}$$

Because $\sigma_H = \delta_H$,

$$\sum_{i=H}^{l-2} s_i w^i + \sum_{i=0}^{H-1} P_i w^{H-1-i} = \sum_{i=H}^{l-2} d_i w_i + \sum_{i=0}^{H-1} P'_i w^{H-1-i}. \tag{7.12}$$

We get

$$\sum_{i=H}^{l-2} (s_i - d_i)\, w^i + \sum_{i=0}^{H-1} (P_i - P'_i)\, w^{H-1-i} = 0. \tag{7.13}$$

Therefore $P_i = P'_i$, where $i = 0, 1, \ldots, H-1$. By reversing the path from $(0, \delta_0)$ to (H, σ_H), we get the exclusive backward path from (H, σ_H) to $(0, \delta_0)$. Thus the request from switch $(0, \sigma_0)$ to switch $(0, \delta_0)$ has to be routed backward using $P_0, P_1, \ldots, P_{H-1}$ ports.

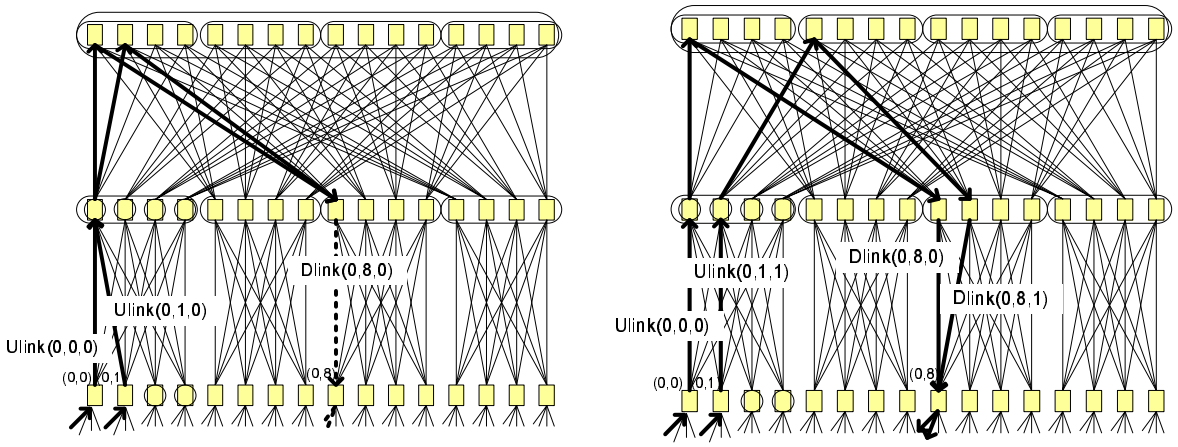
Q.E.D.

7.4 LEVEL-WISE ROUTING ALGORITHM

We find that if the upward path is carefully selected, then the number of conflicts can be reduced. We have shown in Section 7.2 that a request will be routed upward and downward by the same number of upper port at the same level, but may use different switch nodes. A conflict occurs when two requests are routed through same physical path. In conventional adaptive routing, the conflicts occur at downward paths, but are caused by improper selection of upward paths. Figure 7.4 gives an example. Assume that switch (0,0) and switch (0,1) both request a connection to switch (0,8). With local routing information, request from switch (0,0) and switch (0,1) can be routed upward using Ulink$(0,0,0)$ and Ulink$(0,1,0)$. They do not notice conflicts until they are routed back to level 0. In this case, only one of the two requests can pass through to establish a connection. However, if we know the conflict ahead of time, the conflict can be resolved. Since the request from switch (0,0) is forwarded using Ulink$(0,0,0)$, it can be predicted that it must be routed back to switch (0,8) using Dlink$(0,8,0)$. We modify the global routing information to indicate Dlink$(0,8,0)$ is occupied. Thus, when we schedule request from switch (0,1) to switch (0,8), Ulink$(0,1,1)$ and Dlink$(0,8,1)$ will be chosen. By this method, both requests can be granted.

The idea of our centralized routing algorithm is to select upper ports carefully so that the requests can be successfully forwarded both upward and downward at each level with knowledge of the global routing information. A port number is selected only when both the port on the upward port and the port on the downward port are available. For each communication request for a connection between a pair of source node and destination node, we define the data structure shown in Figure 7.5.

In order to grant a communication request, we need to set a path from a source switch to a destination switch. For instance, a connection from source switch $(0,\sigma_0)$ to destination switch $(0,\delta_0)$ is to be established. The common ancestor of both switches is at level H, where $H < l$. In Figure 7.6, a complete path is set, which is switch$(0,\sigma_0) \rightarrow$ Ulink$(0,\sigma_0,P0) \rightarrow$ switch$(1,\sigma_1) \rightarrow$ Ulink$(1,\sigma_1,P_1) \rightarrow \cdots \rightarrow$ switch$(H-1,\sigma_{H-1}) \rightarrow$ Ulink$(H-1,\sigma_{H-1},P_{H-1}) \rightarrow$ switch$(H,\sigma_H) \rightarrow$ Dlink$(H-1,\delta_{H-1},P_{H-1}) \rightarrow$ switch$(H-1,\delta_{H-1}) \rightarrow \cdots \rightarrow$ Dlink$(1,\delta_1,P_1)$ $\rightarrow$ switch$(1,\delta_1) \rightarrow$ Dlink$(0,\delta_0,P_0) \rightarrow$ switch$(0,\delta_0)$.



(a) Routing without local information.

(b) Routing with global information.

Figure 7.4: Routing example.

$(0, \sigma_0)$	$(0, \delta_0)$	P_0	P_1	$\ldots$	P_{l-2}

Figure 7.5: Data structure of a communication request.

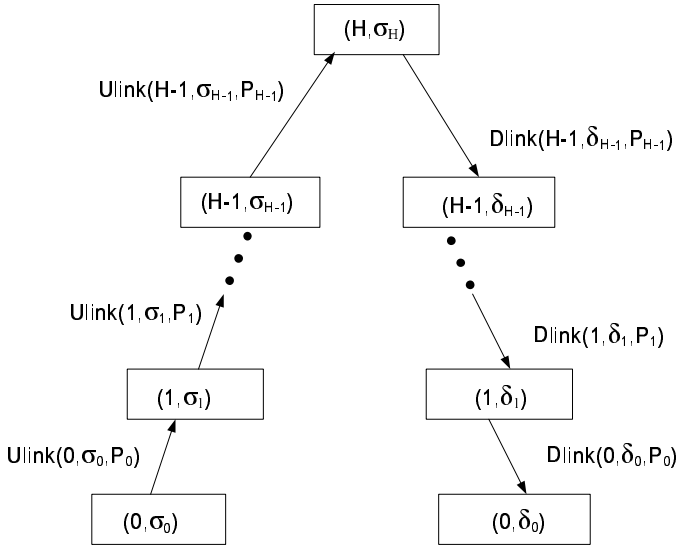


Figure 7.6: Level-Wise scheduling Algorithm.

By representing all links at a given level, h, as a single bit-vector, we can perform simple Boolean operations to determine which port is available on both the upward and downward paths. Specifically, the w bits, Ulink$(h, \sigma_h)[i]$, can be AND'ed with the w-bits Dlink$(h, \delta_h)[i]$, where $i = 0, \ldots, w-1$ to form a w-bit long *available_port* vector that represents the upward ports from the source switch that do not contain a conflict at level h in the fat tree.

All source switches at level h whose *available_port* bit vector contains all '0' values cannot be scheduled. For all schedulable switches, one of the bits in the vector is selected for allocation. For efficiency, we select the first available port and allocate the upward and downward paths. This is performed sequentially on a per source basis. Once level h has been scheduled, a new set of sources for level $h+1$ are created from the set of scheduled sources at level h. The algorithm iterates until all levels are scheduled.

The following is the pseudo-code for scheduling requests that will be routed through level H. We assume switch $(0, \sigma_0)$ is the source switch and switch $(0, \delta_0)$ is the destination switch, where $\sigma_0 = \sum_{i=0}^{l-2} s_i w^i$ and $\delta_0 = \sum_{i=0}^{l-2} d_i w^i$.

The complexity of a conventional algorithm is $O(2l \log_l N)$, while the complexity of our algorithm is $O(l \log_l N)$, where N is the system size and l is the greatest number of levels.

1. for $h = 0$ to $H - 1$ {
2. for a source switch and destination switch pair at level h {
3. avail_links = Ulink$(h, \sigma_h)[0 : w - 1]$ Bit-Wise-AND Dlink$(h, \sigma_h)[0 : w - 1]$
4. if not(avail_links = "000...0") {
5. select i such that avail_links$[i]$ = '1'
6. $P_h = i$
7. Ulink$(h, \sigma_h)[i]$ = '0';
8. Dlink$(h, \delta_h)[i]$ = '0';
   $\sigma_{h+1} = s_{l-2}s_{l-3} \ldots s_{h+1}P_0 \ldots P_h$;
   $\delta_{h+1} = d_{l-2}d_{l-3} \ldots d_{h+1}P_0 \ldots P_h$;
9. }
10. }
11. }

Figure 7.7 gives a fat tree FT$(4, 4, 4)$, which has four levels. Each switch node is a 4×4 crossbar switch. At each level, one of four ports will be selected for upward routing.

It is assumed that a connection is requested from node 3 to node 95. source switch = switch $(0, \sigma_0)$ = switch $(0, s_2s_1s_0)$ = switch (0,000); destination switch = switch $(0, \delta_0)$ = switch $(0, d_2d_1d_0)$ = switch (0,113) The initial request is

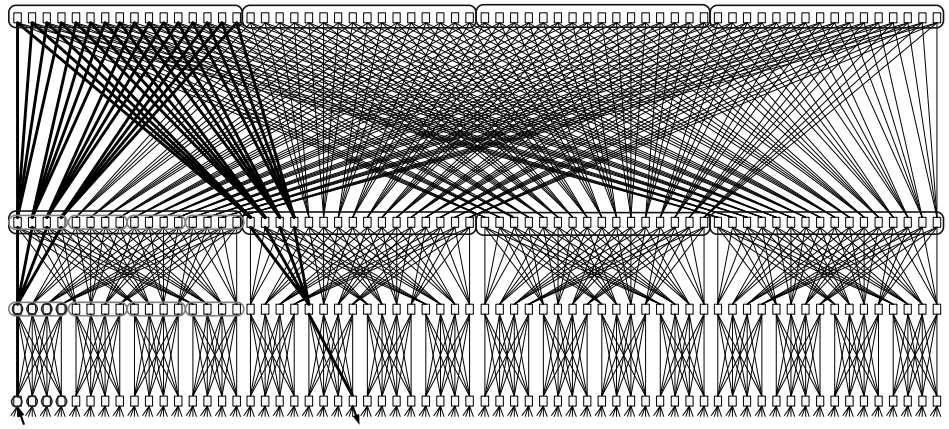
$(0, \sigma_0) = (0, 000)$	$(0, \delta_0) = (0, 113)$	P_0	P_1	...	P_{l-2}

Step 1: select P_0

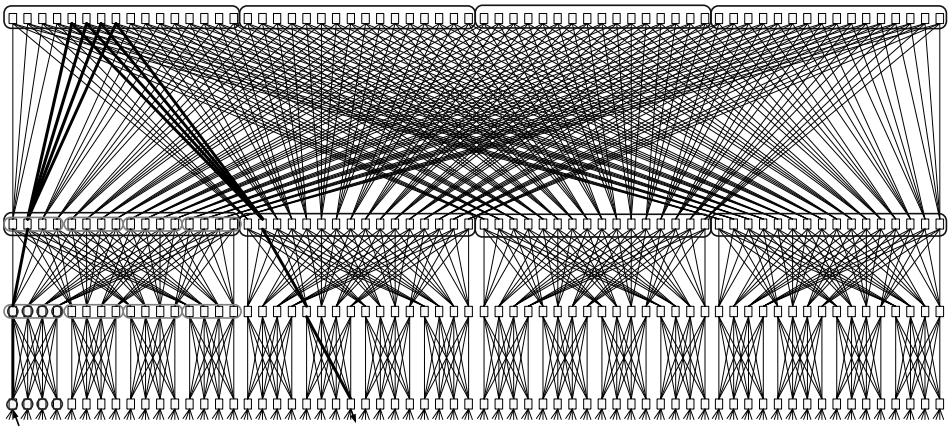
Level = 0
source switch = SW$(0, \sigma_0)$ = SW$(0, 000)$;
destination switch = SW$(0, \delta_0)$ = SW$(0, 113)$.
We assume that Ulink$(0, \sigma_0)[0]$ is '1' and Dlink$(0, \delta_0)[0]$ is '1'. Therefore $P_0 = 0$.

Step 2: select P_1

Level = 1
source switch = SW$(1, \sigma_1)$ = SW$(1, s_2s_1P_0)$ = SW$(1, 000)$
destination switch = SW$(1, \delta_1)$ = SW$(1, d_2d_1P_0)$ = SW$(1, 110)$



(a) P_0 Selection.



(b) P_1 Selection.

Figure 7.7: Level-Wise scheduling example.

We assume that Ulink$(1, \sigma_1)[0]$ is '0' and Dlink$(1, \delta_1)[0]$ is '1', which indicates port 0 is not available. Port 1 is checked.
Let us assume Ulink$(1, \sigma_1)[0]$ is '1' and Dlink$(1, \delta_1)[0]$ is '1', then $P_1 = 1$.

Step 3: select P_2

Level = 2
source switch = SW$(2, \sigma_2)$ = SW$(2, s_2P_0P_1)$ = SW$(2, 001)$
destination switch = SW$(2, \delta_2)$ = SW$(2, d_2P_0P_1)$ = SW$(2, 101)$

We assume that Ulink$(2, \sigma_2)[0]$ is '1' and Dlink$(2, \delta_2)[0]$ is '1'
Therefore, $P_2 = 0$

$(0, \sigma_0) = (0, 000)$	$(0, \delta_0) = (0, 113)$	$P_0 = 0$	$P_1 = 1$	$P_2 = 0$

All ports are allocated successfully so the request from switch (0,000) to switch (0,113) is granted.

7.5 SIMULATION RESULTS

The purpose of the simulation is to compare the performance of our Level-Wise scheduling algorithm with local scheduling algorithm. We have performed system level simulation for randomly generated communication permutation based on two-level, three-level and four level fat-tree interconnected networks.

The experiments are based on SystemC based simulator to simulate scheduling procedures. We have built switch nodes and connected them in fat-tree topology. Each switch node has bidirectional input and bidirectional output. Network control signals, e.g. communication grants and requests, are passed through each switch node in parallel.

We generate random communication permutations as requested connections. Those communication requests are assigned as inputs of each source switch node. The requests are scheduled by our Level-Wised scheduler and general adaptive fat-tree scheduler with local routing information. If a requested connection is successfully established, the request will be forwarded to the destination node. By checking the control signals received at destination

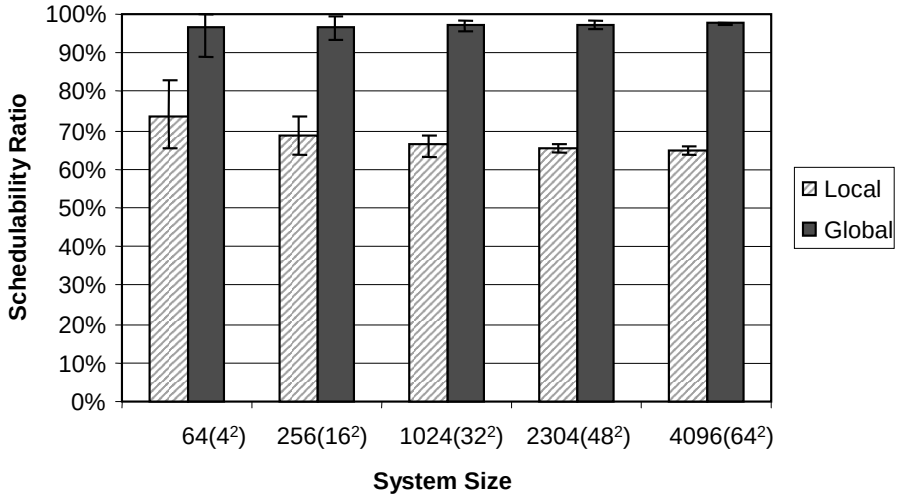


Figure 7.8: Level-Wise scheduling Algorithm. (Two-level fat-tree interconnection network.)

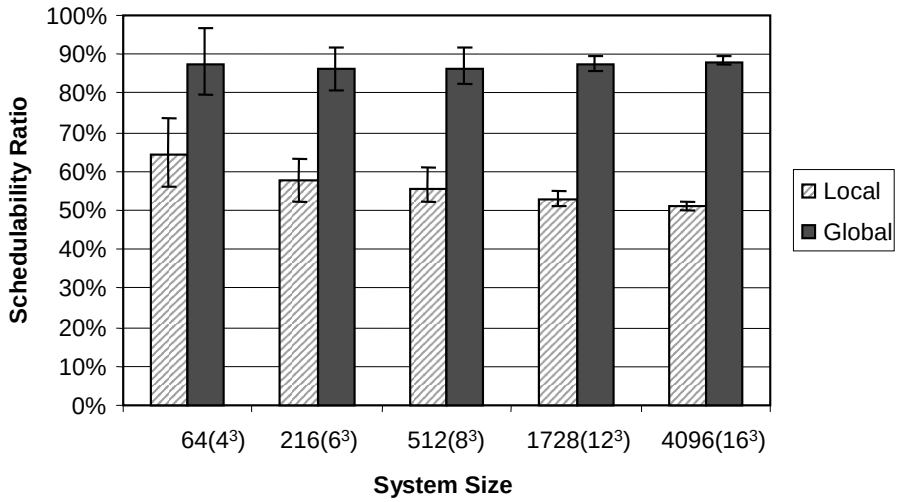


Figure 7.9: Level-Wise scheduling Algorithm. (Three-level fat-tree interconnection network.)

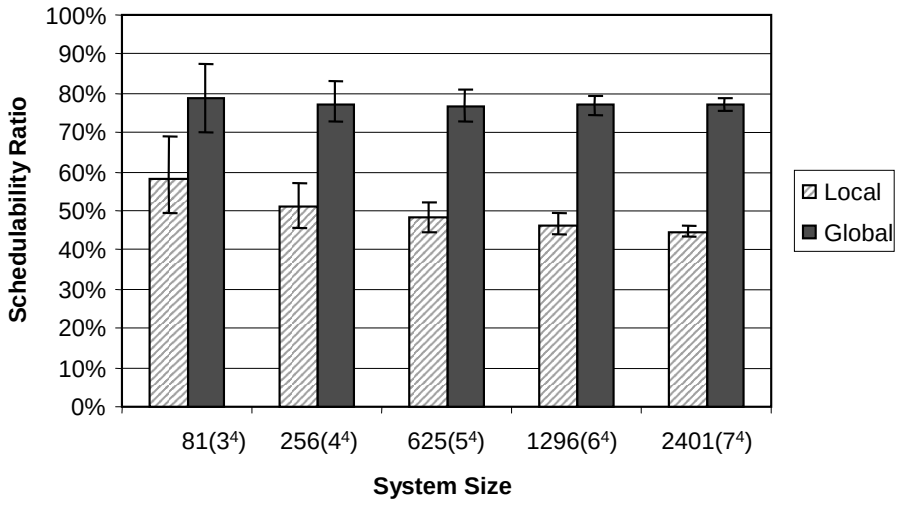


Figure 7.10: Level-Wise scheduling Algorithm. (Four-level fat-tree interconnection network.)

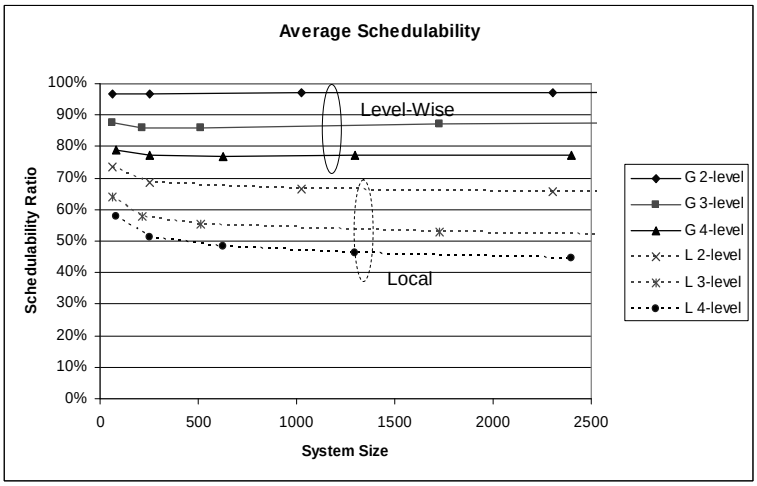


Figure 7.11: Schedulability comparison based level-Wise scheduling Algorithm.

nodes, we are able to compute the number of scheduled connections. We generate a set of 100 random permutations for each test point. We use schedulability ratio to evaluate the scheduling capability, which is defined as the number of successful connections divided by the number of total requests.

Figure 7.12, 7.13, and 7.10 compares the schedulability ratio in two-level, three-level and four-level fat-tree communication networks. Each bar represents the average schedulability ratio in our experiments. The upper line and lower line at the top of each bar shows the maximum and minimum schedulability ratio respectively.

It can be observed that the Level-Wise scheduling method provides higher schedulability ratio than the convention local scheduler. The deviation of the schedulability ratio become less as the system size increases. This is because in either scheduling algorithms, the selection of upper port for upward routing is flexible and the path for downward routing is fixed. In the large-scaled system, communication requests has more upper ports to select when they are forwarded upward, which increases the possibility for successful routing. The minimum schedulability ratio of the Level-Wise scheduler is higher than the maximum schedulability ratio of the convention scheduler. That is to say the Level-Wise scheduler is able to schedule more connections even in its worst case than the convention scheduler.

It can also be observed that the schedulability ratio decrease as the number of levels increase as shown in Figure 7.11 because for a high-level system, more levels need to be adjusted to configure a connection. A communication request may be successfully routed for most levels, but a failure routing in one level will destroy all previous routing for this request. However, our scheduler using global information always outperforms the convention scheduler using local information. As the system size increases, the improvement becomes more significant. In a network more than 500 communication nodes, the improvement is over 30%. Note that only systems of size w^l are used for l-level systems.

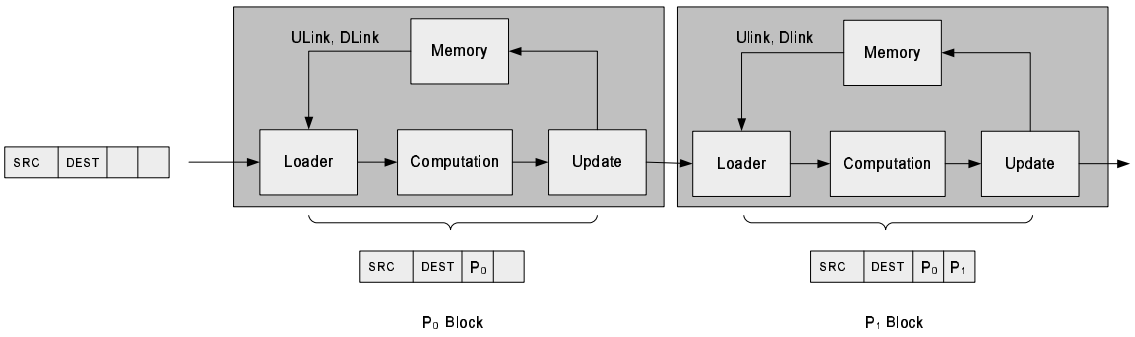


Figure 7.12: Level-Wise scheduling Algorithm.

7.6 LEVEL-WISE SCHEDULING HARDWARE ARCHITECTURE

Our scheduling algorithm can be implemented by software or hardware. We give pipelined hardware implementation architecture that is to be used as a centralized scheduler. Our hardware design is targeted on FPGA.

In order to introduce the hardware design more clearly, we describe the scheduling procedure from a system's level. With the knowledge of the system, the detailed hardware function blocks are illustrated.

Figure 7.12 shows a hardware system for scheduling a three-level fat-tree interconnection network. For a three-level fat-tree, each communication request associates with no more than two upward ports, which are P_0, and P_1. Two main function blocks, P_0 Block and P_1 Block process the value of P_0 and P_1 individually. When one request comes, P_0 Block computes the value of P_0 and the value will be used in P_1 Block to compute the value of P_1. While P_1 Block processing Request 1, the P_0 Block can process in pipe for second request, Request 2.

P_0 Block and P_1 Block have similar hardware structure, so we just pick P_0 Block to describe the hardware design in detail. It consists of three function blocks, 'Load', 'Compute', and 'Update'. In load stage, the source switch and destination switch are computed and current Ulink and Dlink vector are read from the two memories. Compute stage is

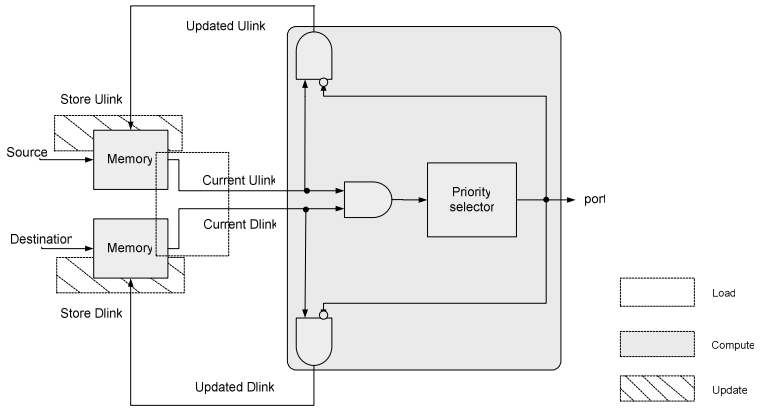


Figure 7.13: Level-Wise scheduling Algorithm.

pure combinational logic, which computes valid port number and updated Ulink and Dlink vectors. In update stage, the updated Ulink and Dlink vectors are written back to the two memories. Figure 7.13 illustrates the hardware architecture of P_0 block. We use white, grey, and shaded blocks to indicate 'Load', 'Compute', and 'Update' function blocks individually. Two memories are used to store Ulink and Dlink vectors. During a scheduling procedure, the current Ulink and Dlink vectors are output corresponding to the memory address, which are source switch and destination switch. A port i can be picked up only when both Ulink$(h, \sigma_h)[i]$ and Dlink$(h, \delta_h)[i]$ equal '1'. We first combine Ulink and Dlink vectors by logic 'and', then select one available port using a priority selector.

Based on the pipeline hardware architecture, we have developed synthesizable hardware components and target our design on Altera Stratix II FPGA. Hardware for scheduling three-level fat-tree interconnection network has been designed, with system size varying from 64 to 4096. Table 7.1 gives the post place and route synthesis results.

The result shows that for a communication system with 4096 communication nodes, our centralized scheduler is able to schedule an individual communication request within 20ns. Using less than 40 μs, 4096 communication requests can be scheduled.

Table 7.1: Performance evaluation (Design targeting on Altera Stratix II FPGA)

System Size	Processing Time	
	Schedule one request	schedule all requests
64 (4×4 switch)	15 ns	480 ns
512 (8×8 switch)	17 ns	4 352 ns
4096 (16×16 switch)	19 ns	38 912 ns

7.7 CONCLUSION

By using global routing information, our scheduler is capable of configuring a fat-tree interconnected network in a close-to-optimal fashion. This is especially beneficial to setup long-lived connections. We have developed pipelined hardware architecture to evaluate complexity and processing capability of the scheduler. Our simulation results show the Level-Wise algorithm improves schedulability efficiently without adding extra complexity compared with a convention scheduler.

8.0 NETWORK SIMULATION FRAMEWORK

This chapter introduces a framework for the design, synthesis and cycle-accurate simulation for parallel computing networks of 128+ processors. In order to accurately characterize the network, we present a bottom-up design methodology in which each of the components are designed using a hardware description language and synthesized to an FPGA for performance estimation of the final ASIC implementation. The components are then integrated to form a parallel computing network and simulated using a cycle-accurate simulator with network traffic described by command files. This enabled us to simulate various switching techniques. Our results show that this hardware design, synthesis, and cycle-accurate simulation methodology provides a useful method for evaluating design tradeoffs in parallel networks.

In Section 8.2, we provide background information about the network simulation. A description of our design, synthesis, and simulation methodology is described in Section 8.3, along with a description of each of the components in our design and simulation framework. In Section 8.4, we show how these different components can be assembled to form four different types of networks, and how they can be simulated with cycle-level accuracy. The validation of our simulations are described in Section 8.5, and conclusions are offered in Section 8.6.

8.1 INTRODUCTION

A network's design is not only dependent on the topology and routing algorithms but is also dependent on the design of each of the different components. For clusters, a network

includes the network interface cards, the cabling, the switches and the overall topology. The performance of the entire system is also dependent on the interactions of these components when they are being used by different traffic patterns. In order to gain insight into both of these areas we introduce a modular approach that decomposes the network into its individual hardware components for accurate characterization and we introduce two methods of interconnecting these components to simulate the dynamic behavior of large parallel systems.

In this chapter, we present a unified framework and bottom-up methodology for hardware design, synthesis, and cycle-accurate simulation of parallel computing networks. A major objective of this effort is to build a modular design and simulation framework in which components can easily be assembled and modified to build different systems.

For evaluation, we utilized a cycle-accurate hardware simulator, available for ASIC and FPGA hardware design, providing the ability to inspect different signals down to the nanosecond level of detail. The modular simulator Simple Scalar [77], which has been built for computer architecture research, and Network Simulator [78], for network simulation, are similar examples. Using this methodology in our simulation, each of the components is designed using a hardware description language and synthesized to an FPGA for performance estimation of the final ASIC implementation. These components are then integrated to form an entire network of N processors capable of sending and receiving data as specified in command files. This uniform design and simulation framework enables direct comparison of various switching techniques for parallel computing networks.

8.2 BACKGROUND

In order to facilitate new designs, a variety of simulators have been created. As early as 1976, CEGRELL built a simulation model to study a full-duplex message switched computer network [79]. A lot of research has been performed on building specific simulation models to evaluate network performance. As indicated by Mars [80], four general approaches are normally used to simulate a communication network: using a general purpose simulation language (e.g. SIMSCRIPT [81]), using a communication oriented simulation language (e.g.

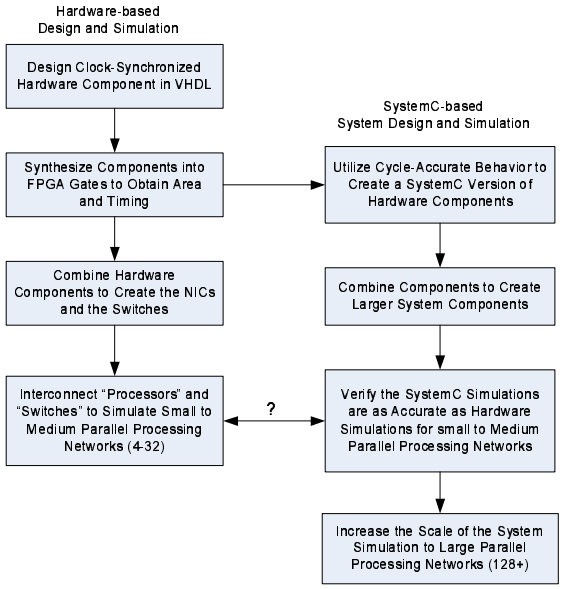


Figure 8.1: Design flow methodology to create cycle accurate simulations for large system sizes using VHDL and SystemC.

OPNET [82]), using a communication oriented simulator (e.g. BONES [83]), and using a general purpose language (e.g. C/C++). Rexford and his colleagues [84] presented an object-oriented discrete-event simulation for evaluating network designs. Liu and Dickey [85] studied buffered and un-buffered switch networks by changing the configuration of the buffers in their simulation. Gorton, Kerirdge and Jervis built a simulator, called Occam, to simulate microprocessor system at component level [86].

8.3 DESIGN AND SIMULATION METHODOLOGY

The objective of our methodology is to provide a rigorous design flow for high-performance parallel processing networks that scale to hundreds or even thousands of nodes. This presents design and simulation problems as simulators are typically software-based while ASICs are utilize hardware description languages (e.g., VHDL, Verilog) that requires

a complex set of design and simulation tools. By combining these design and simulation methodologies, we ensure that the simulation components have the exact behavior as their ASICs counterpart. Figure 8.1 shows the combined hardware and software design flows. The left column shows a traditional hardware design and simulation flow for ASICs and FPGAs while the right column shows the transformation of the results from the hardware design flow into software components. Integration and interconnection of these software components can form larger components and/or systems. Verification between the hardware and software systems is possible for small to medium scaled networks.

For accuracy, we have designed and implemented our components using the VHDL hardware description language to prove that our components represent real hardware. To gain nano-second level performance data, we have synthesized our VHDL into FPGA gates using Mentor Graphics Precision Synthesis software [87]. This enables the extraction of both latency (i.e., cycles of delay for the first result) and bandwidth (i.e. number of results per cycle under steady state conditions). The synthesis tool performs a detailed timing analysis and reports a maximum clock frequency. From our functional simulations, we can determine the number of cycles required for any given operation. By multiplying this cycle count by the nanosecond duration of the cycle, which is one over the maximum frequency, the component latency can be determined. The end design is expected to target ASIC technology, but FPGA timing results can be more easily obtained and compared with published results. We conservatively estimate that the ASIC performance will be five times faster than the FPGA results. We could incorporate ASIC synthesis tools into this flow to improve accuracy to the sub-nanosecond level but this fine-tuning is not necessary for even moderately different networks as long as all systems utilize the same hardware estimations.

The hardware behavior and performance is then used to create an identical module in SystemC, a C++ variant that enables cycle-accurate simulations. These SystemC modules can be interconnected and compiled to produce an executable that simulates the behavior of the entire network. Thus, this framework provides a methodology for designing entire parallel networks that are as accurate as hardware simulations but enable large systems to simulated in reasonable amounts of time on a single workstation.

SystemC is a C++ based hardware design language that was developed to promote system-level simulation and to enable hardware-software cosimulation [88, 89]. Fundamentally, SystemC is a set of parameterized template classes built in C++ that allow the creation of hardware structures available in other languages such as bit-vectors, processes, and ports. Like other hardware languages, such as VHDL and Verilog, it is possible to describe a SystemC design behaviorally, at the register-transfer level, and structurally. The advantage of SystemC is most highly visible in the fact that it essentially C++ code. As a result, SystemC designs, along with their corresponding test benches, may be compiled directly into a software binary that becomes a custom simulator for that particular hardware design. Similarly, for system-level simulations designed in SystemC, combining software and hardware portions becomes much easier as they can be combined and built using a single program. For traditional hardware simulation techniques such as using ModelSim [90] for VHDL or Verilog, a foreign language interface is required to communicate between hardware and software components. The most important advantage of SystemC for our simulation environment is its increase in capacity over more traditional hardware simulation methods. Because ModelSim must be able to simulate every VHDL construct, even those rarely used, it incurs significant overhead. For the equivalent SystemC simulation, a custom simulator is built and only the components required for the application are incorporated into the simulator. From our experience, this results in accelerated performance by a factor of three and results in a factor of five for memory utilization. In fact, we found that our VHDL simulations using ModelSim only scaled to 32 processors while our SystemC simulations scaled to over 128.

In order to accurately build and simulate a high-performance multi-processor network, the network interface controller (NIC) hardware and the switch element(s) must all be designed in hardware for maximum performance. To simulate the entire system, the processing elements, the wires and the topology must be accurately modeled but do not need to be designed using a hardware description language. However, to validate our designs, we built a 32 processor system entirely in VHDL and then built an equivalent system in SystemC using equivalent components. In the next section, we show the system-level results but in this section, we focus on the fundamental components that we created and reused throughout the

different systems that we built. The Processing Element component reads data transmission commands from a file and sends data into the NIC. The Processing Element component also receives data from the NIC and records it to a different file with a timestamp. The NIC, however, was designed in hardware using three different components: a Single-Wide Data Queue component, a N-Wide Data Queue component and custom control logic. The two different types of data queues are described in more detail in this section. The Wire component emulated the behavior of a high-speed network cable. The switch is comprised of a Scheduler component and a Switch Fabric component. The Scheduler determines the configuration of the switch and as a result, its performance and design is central to the network's performance. Thus, the Schedule component was designed in hardware. The Switch Fabric can be an analog, digital or optical device and, as such, only its behavior is described. All buffers within the switch were modeled using the data queue components. The reminder of this section describes each of the individual components while the next section describes different systems that we constructed from these components.

8.3.1 The process element component

A significant portion of a communication's delay is in software overhead and in moving data from the processor, or from memory, to the network interface card. The literature supports the benefit of innovative approaches in this area. However, this paper focuses on the performance of the network and does not consider the delays associated with the processor/memory to NIC interface. We are doing this for two reasons. First, the only modifiable components within a processing node in a cluster are the network interface cards and the software executing on the processor, with a fixed processor-to-NIC interface. Second, the network design and the processor interface are not tightly coupled. Improvement on the processor interface will help all networks and improvements on the network will benefit all types of processor interfaces. Thus, we virtualize the processor as an outgoing queue that contains data to be sent out onto the network, and as an incoming queue that receives packets from the network.

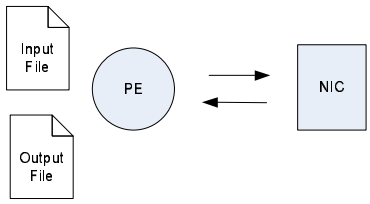
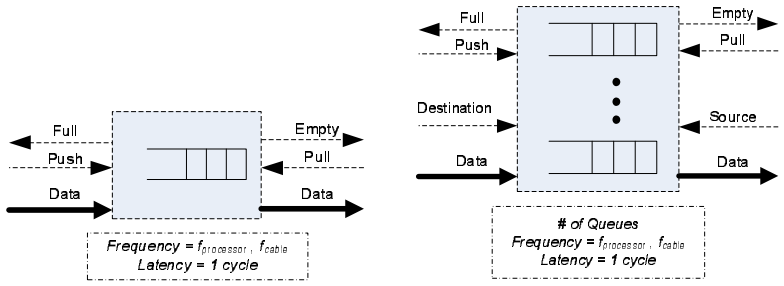


Figure 8.2: Processing element components.

Each processor has its own input file that contains a number of predefined commands, shown in Figure 8.2. The command send tells a processor element (PE) to generate data with a specific message size and destination. The command wait emulates a period in which the PE is performing computation and thus, no traffic is generated. In addition to these basic commands, advanced commands can be grouped to perform more complex MPI functions, like broadcast, blocking send, blocking receive, and barrier, among others. The amount of data that is sent is described in the input file, but the actual data that is sent is not important to the network operation, as the network does not inspect the data payload of the packets. For debugging purposes, however, the payload of the packet is used to send the source and a timestamp the packet was created. At the destination PE, this information along with its arrival time is stored in the output file for post processing and performance summary. This processor model allows us to test a variety of traffic models by simply creating a set of input files.

8.3.2 Data queues

One of the fundamental components in a network is the data queue. Anywhere data is being sent or received, there is a need to buffer data. Functionally, a queue receives a stream of data and outputs the stream in the same order. We have created two different data queues, a Single Queue and an non-blocking N-Queue, shown in Figure 8.3. The Single Queue is simply a first-in, first-out queue, while the N-Queue is a single component that represents N different queues grouped together with a single write port and a single read port.



(a) Parallel to serial.

(b) High speed to serial.

Figure 8.3: The single queue and the N-queue components.

The *Full* and *Empty* status lines indicate the availability of the queue for writing and reading, respectively. For the Single Queue, these status lines are a single bit wide and for the N-Queue they are N-bits wide to indicate the status of each of the N internal queues. Writing data into the Single Queue simply requires holding the *Push* signal high for a single cycle, but for the N-Queue the *Destination* queue must also be specified. Likewise, holding the *Pull* signal high for a single cycle will read data from the Single Queue, but with the N-Queue the *Source* must also be specified.

The performance of queue can also vary. If the *Pull* signal is active (i.e., a '1'), the queue outputs a data value every cycle. The frequency of data movement into or out of the queue is one word per cycle. This frequency along with the width of the queue determines its bandwidth. The latency of the queue is the amount of time between placing a data value in an empty queue and the time it can be removed.

The Single Queue was already designed within the Mentor Graphics HDL Designer system in their ModuleWare library [91]. Similarly, both major FPGA manufactures, Xilinx [92] and Altera [93], have wizards for configuring FIFOs that automatically generate synthesizable hardware components. The Simple Queue has a single clock cycle of latency through the queue, with the cycle frequency of 108MHz for the Altera FPGA, EP1S25F1020C-5. Thus, its throughput is 108 million words per second where the width of the queue is the word size that can be arbitrarily configured. The latency is one cycle or 1/108MHz = 9.2ns.

The N-Queue can be designed in numerous ways, depending on the objective sought. The simplest implementation is to replicate the Single Queue N times and multiplex the *Head* and *Tail* of the queues using the Source and Destination as select lines, respectively. While this is appealing from a rapid design perspective, it suffers from inefficiency, as N dual-ported RAMs and N comparators are needed. We observe that during any given instance at most one queue will have data placed into it and at most one queue will have data retrieved from it. The same queue can have both read and write access simultaneously but this means that only a one dual-ported memory is needed to buffer the packet data. There will need to be N head pointers and N tail pointers to addresses in data RAM. The problem is keeping track of the head and tail pointers for each of the N internal queues, as well as updating the *Full* and *Empty* status lines.

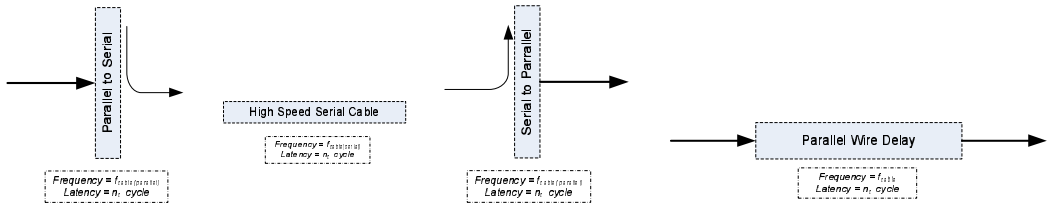
For our design, we implemented the *Head* and *Tail* pointers using two small register files that have three address ports. For a *Pull* operation, the *Head* pointer is used to specify the address in the data RAM for reading, and for a *Push* operation, the *Tail* pointer is used to specify the write address in the RAM. The second port is used to write back the incremented pointer after the read, or write, is performed. The third port is used to update the *Full* and *Empty* flags. On a *Pull* operation, the head-of-queue is incremented, if the *Head* and the *Tail* pointers are the same, then the queue is *Full*. On a Push operation, if the incremented *Tail* and *Head* pointers are the same, then the queue is *Empty*. Thus, each register file must have two read ports and one write port. We obtained hardware area and performance results by synthesizing our VHDL to an FPGA. There is negligible decrease in performance as it scales to 128 queues but a proportional increase in circuit size (i.e. logic cells) which is due to the 3-ported register file. ASIC results are expected to be five to ten times faster.

8.3.3 Wires

The physical layer interconnection of a system can have a drastic and dynamic impact on the entire system [94]. Early in the system design, the characteristics of the communication channels are specified in general terms. This may include the functionality, latency, bandwidth, and bit-error rate of each network link.

Table 8.1: N-Queue hardware synthesis and performance results N=4 to 128, Width=64 bits, FPGA target: Altera EP1S25F1020C-5

N	4	8	16	32	64	128
Logic Cells	361 (1.4%)	480 (1.9%)	1,439 (5.6%)	1,988 (7.8%)	3,939 (15.4%)	8,010 (31.2%)
Memory (bits)	16,386 (0.8%)	32,768 (1.7%)	65,536 (3.4%)	131,072 (6.7%)	262,144 (13.5%)	524,288 (27.0%)
Clock Constraint (MHz)	78	88	69	67	63	59
Throughput (Gbps)	4.9	5.6	4.4	4.3	4.0	3.8
Latency (ns)	13	11	14	15	16	17



(a) Parallel to serial. (b) High speed to serial. (c) Serial to parallel. (d) Parallel wires.

Figure 8.4: Wire delay models.

In complex systems, these characteristics alone can significantly alter a system's performance and guide the underlying system design. Optical switches may combine these performance characteristics with unique topologies such as multicasting and time/wavelength division multiplexing. In order to design a system effectively, these communication characteristics must be simulated along with the entire system.

At a high-level of abstraction, physical interconnections can be modeled as parallel wires that contain a specific delay. This is easily achieved using the VHDL command `A<=B after 5ns`, which assigns the value of B to the output A after a delay of 5ns. This command can be used for busses as well as wires, but causes a latency delay and a bandwidth of the same value. For example, changing B from low to high and then back to low within a 5ns period will not cause a corresponding change in A. However, using the VHDL statement: `A <= transport B after 5ns` will enable changes smaller than 5ns to be seen on A. When the source of the signal is generated by a clock edge, or is regulated by some other portion of the circuit, then the simple delay is sufficient for modeling and more complex mechanisms are not required.

For our simulations, we have defined four components as shown in Figure 8.4. The First component is a Parallel-to-Serial converter, the second is a High-Speed Serial Cable, the third is a Serial-to-Parallel converter, and the last is a Parallel Wire. For the Parallel-to-Serial component, there will be a one-clock cycle delay associated with this component at the clock cycle frequency of the sender. The High Speed Serial Cable has a latency that is

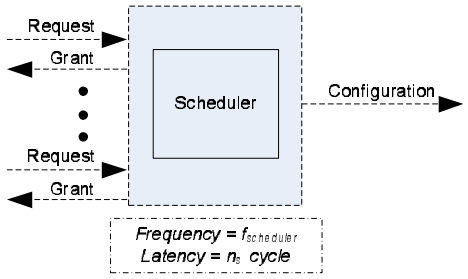


Figure 8.5: The scheduler.

proportional to the length of the cable being simulated and can be conservatively estimated as one to two ns per foot. The bandwidth for this cable must also be specified, as this will determine its throughput. The Serial-to-Parallel converter must wait until all of its bits have been serially shifted into the register, thus, there is a latency associated with this component. This latency is its Width / Serial Frequency, which must be equal to its Parallel Frequency. Lastly, the Parallel Wire component is used to define delays that are within a chip. It should be noted that the Parallel Wire component can also be used to simulate a sequence of Parallel to Serial, High Speed Serial and Serial to Parallel components.

8.3.4 Network scheduler

One of the critical components of a network is its arbitration logic. If multiple processors send data to a single destination, there will be a conflict within the network that needs to be resolved. Within a packet switched network, this would be seen as multiple packets in different input ports that have the same output port. For circuit switching, multiple NICs would request a circuit to the same destination. Irrespective of the network type, there must be some arbitration logic that determines which processor, or port, gets priority.

To handle arbitration, we created a component called a Scheduler that receives up to N requests for N destinations as shown in Figure 8.5. Each *Request* input is an N-bit bit-vector, which specifies the destinations to which it needs to send the data. For PE_j, if *Request[i]* = '1' then PE_j has data that it needs to send to PE_i. The output of the Scheduler is a *Grant*

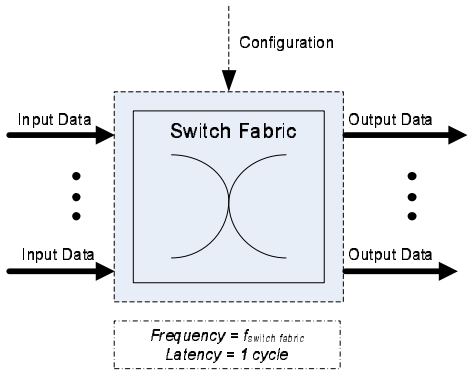


Figure 8.6: Switch Fabric.

signal for each of the *N Requests*, in which each PE is granted at most one destination that was requested. A *Configuration* is output from the Scheduler to the Switch Fabric indicating its configuration.

8.3.5 Switch fabric

Switching fabrics have been around since the early days of parallel computing with a mature research field in multi-stage interconnection networks. In this paper, we have separated the control portion of the switch, which we refer to as the scheduler, from the data path portion of the switch. The switch fabric can be a single crossbar, a multistage interconnect switch, or any kind of topology as shown in Figure 8.6.

Depending on the type of technology used to create the switch, the configuration time and the propagation time, i.e. the configuration and data latency, respectively, can change by orders of magnitude. Digital switches have faster switching latency, but have a longer propagation delay and a lower per-pin throughput than low voltage differential signal (LVDS) switches and optical switches.

8.4 SYSTEM SIMULATION

This section illustrates the design methodology for large systems using the proposed design and simulation framework. In the prior section, we described the individual components that were created using a hardware description language and characterized by their FPGA performance. We examine the characteristics of three networks and show how different networks, with different characteristics, can be compared using the common set of components and simulation framework. The three different types of networks we created are packet switching, circuit switching, and predictive circuit switching. In this section, we show the various parameters that can be set for the different components.

The behavior of each module is predefined but the performance can be modified. Each module is given a *frequency* and a *latency* parameter. Recall that the inverse of the frequency of a hardware device is duration of one clock cycle and that this duration is determined by the target technology. As the density of transistors increases, the clock frequency also increases. This parameter is therefore technology dependent. The latency of a particular component is specified as the number of cycles required to achieve the result. This parameter is design-dependent and can be derived from the architecture of the component. The internal storage of a component, if applicable, is also design-dependent. Thus, by specifying the frequency, the latency and the buffer size of each component we can characterize an entire system. Many components utilize the same frequency as will be shown.

8.4.1 Wormhole switching

Wormhole switching networks decompose all communication into small point-to-point messages that are routed through the network independently. At the destination, the original message is reassembled from the individual packets.

To simulate a wormhole switching network we represent the processor as a data-sender and as a data-receiver and do not consider the overhead associated with the processor interface or the creation of network packets. As such, we expect that there will be an additional latency, for a "real" network, that is not considered here. However, our goal was to be able

to compare and contrast different networks and, as such, the processor and bus interface circuitry would be the same.

The network interface card/controller is shown in Figure 8.7 as two Simple Queues, one for output traffic and one for inbound traffic. For clarity, only one NIC for sending and receiving data is shown. In our simulation, data can be sent and received by NICs simultaneously. A small amount of control logic was added to the NIC to handle backpressure, not shown in the figure. For the switch, we implement an input buffered switch and a single scheduler to perform the routing/arbitration. We simulate this using an N-Queue component for inbound traffic, a Scheduler component for routing/arbitration and a Switch Fabric component for the crossbar. Data coming from the switch into the NIC is buffered into a Simple Queue and written to a file with a time stamp by the processing element. Each worm was created with using 64-bit words, a one word header, a one word flit, a ten flit payload, and a one word tail.

By using input buffering, we enabled the scheduler to improve the switch utilization, since it has knowledge of all destinations for each of the N ports. As described earlier, we have implemented these N-Queues in hardware and have shown that head-of-line blocking can be avoided even for a large N by using hardware, as long as there is only one word written and one word read per cycle. This assumption is realistic for a switch, as each port has a single cable receiving inputs and a single switch fabric interface. This, however, is not true for all designs and must be taken into consideration. The Scheduler component receives an N-bit vector from each N-Queue specifying which of its internal queues have data. It then allocates the bandwidth using the round-robin priority scheme described in Section 8.3 but schedules all destination ports within a cycle. This enables a single-cycle allocation of bandwidth and enables out-of-order routing from each port, both of which increase network utilization. Other schemes can be implemented by simply changing the design of the Scheduler and re-running the traffic traces.

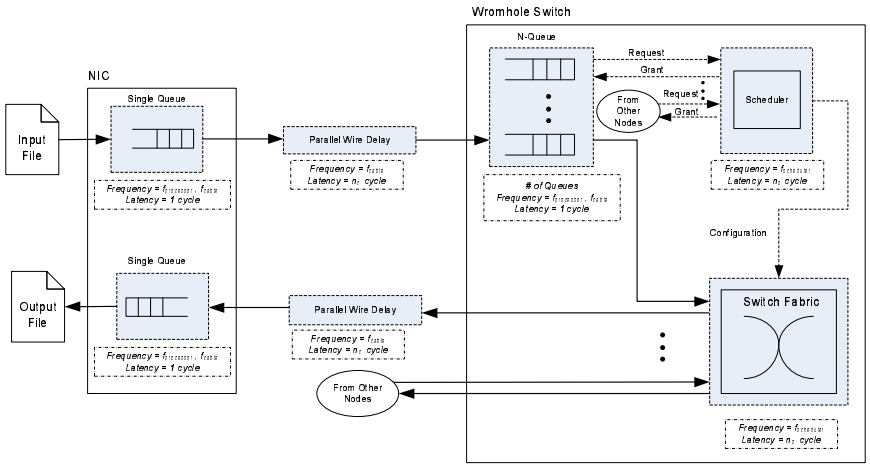


Figure 8.7: Wormhole switching network..

8.4.2 Circuit switching

Circuit switching utilizes the same components as wormhole routing even though these are very different networks. Surprisingly, the major difference is the location of the buffers and the corresponding distance between these buffers and the schedule, shown in Figure 8.8. For wormhole routing, the N-Queue was next to the switch fabric, but in circuit switching, this buffer is located within the NIC. Thus, each time a packet needs to be sent, a request must first be sent to the scheduler, the scheduler must determine if the request can be granted, the circuit is established and an acknowledgement is sent to the NIC. The circuit is maintained until the NIC's buffer is empty for that particular source-destination connection.

Some switch fabrics, like all optical switches, cannot buffer data and thus, circuit switching is required. By having a central scheduler that has knowledge of all of the data that needs to be sent in any given cycle, there is a greater chance to improve network utilization. For large networks, a wormhole routing switch only sees data that are in its queues while a circuit switch with centralized scheduler has complete knowledge of all pending traffic. For

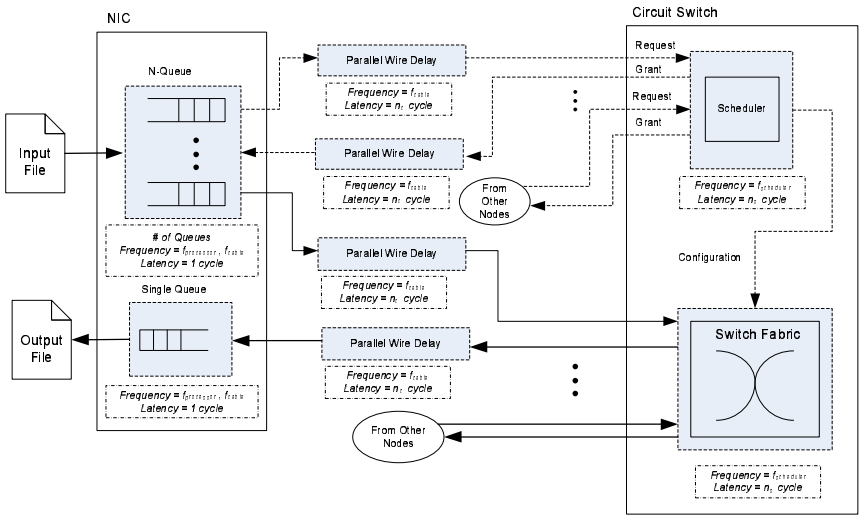


Figure 8.8: Circuit switching network.

networks with multiple stages, the wormhole routed switch will only have local knowledge and not global knowledge of the pending data traffic.

8.4.3 Predictive circuit switching

In order to avoid the overhead of circuit switching, we introduce predictive circuit switching, shown in Figure 8.9. In this type of network, the circuits are setup before they are requested. This concept is analogous to cache prediction, but rather than fetching data before it is requested, predictive circuit switching configures the network before it is needed. If the network predicts accurately, namely a "hit", then there is no setup latency and the network appears as if each source is directly connected to its destination. Circuit switching removes the buffers from the switch and reduces its latency of a switch to that of a small segment of cable because data can stay in the optical or analog domain. By predicting the next connection, the scheduler delay is hidden because it is pre-computed. The switch configuration still exists, but is minimal for LVDS switch elements and other similar technologies.

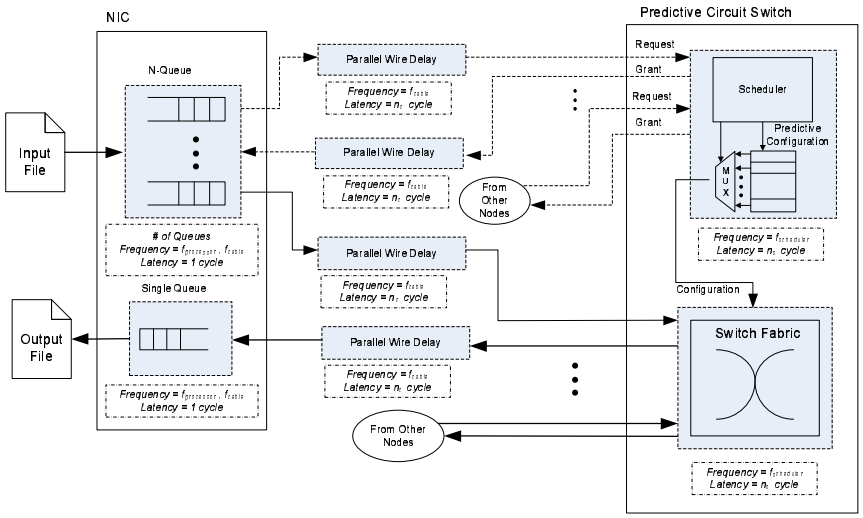
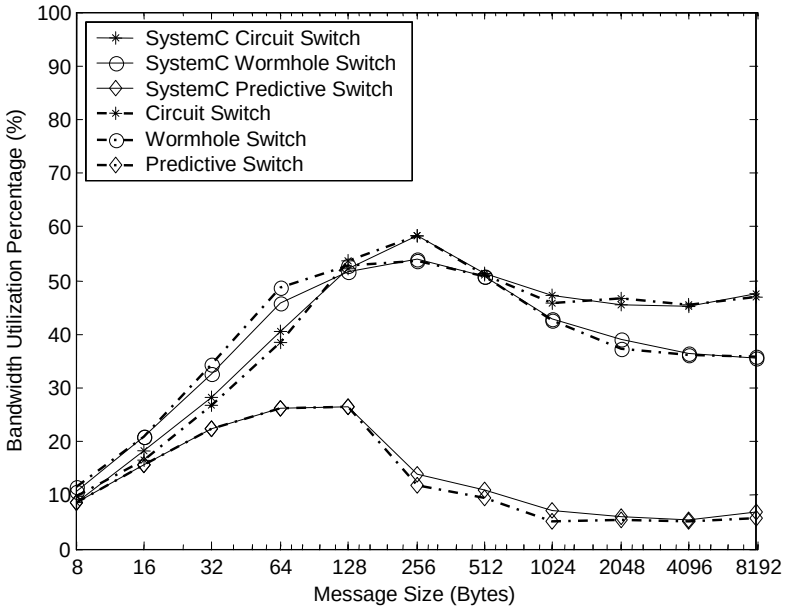


Figure 8.9: Predictive switching network.

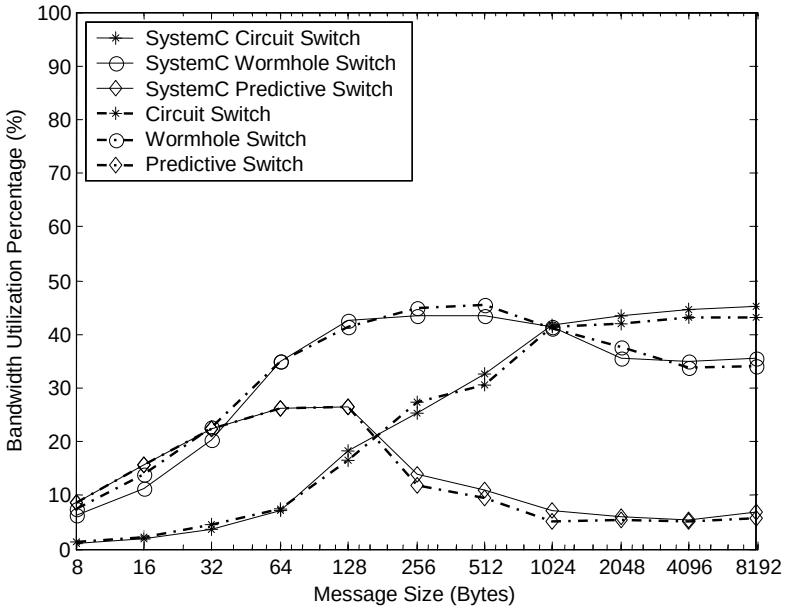
However, when the switch predicts incorrectly, there is a miss penalty that can be substantial. If the network supports preemption, the penalty can be little more than that of circuit switching, but if the predictive circuit switch has a complete, predefined set of configurations, then an unpredicted communication may have to wait for a few communication cycles before it can sent its data. This paper introduces the concept of predictive circuit switching, affirms its benefits during predictable traffic, and examines its drawbacks during unpredictable traffic. For this initial discussion and simulation, we use a round-robin prediction method that cycles a fixed set of destinations. We have demonstrated through simulation that this scheme is better than packet and circuit switching when there is a high degree of predictability, and that this scheme has a high cost for missed predictions.

8.5 SCALING FROM 32 TO 128 PROCESSORS USING SYSTEMC

Our experiments are based on wormhole switching network, circuit switching network, and predictive switching network. All parameters in our simulation are configurable and in the

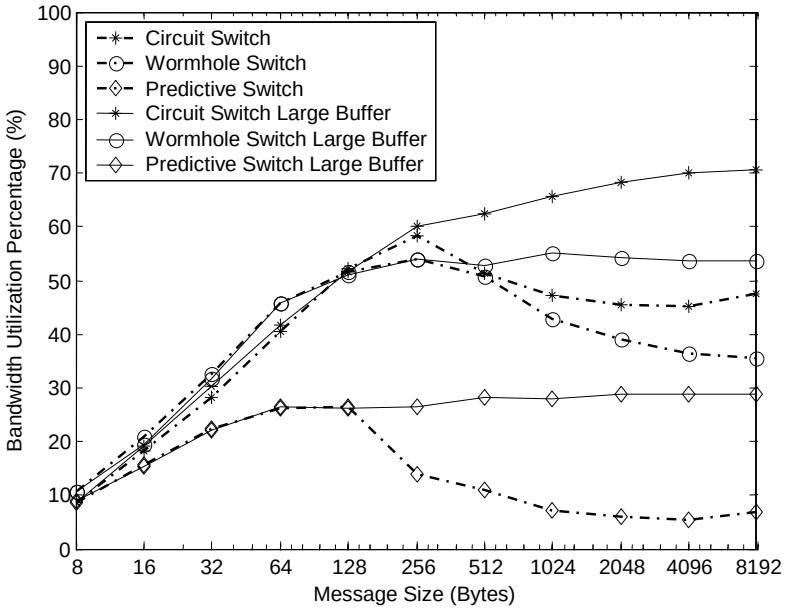


(a) 10 foot cable.

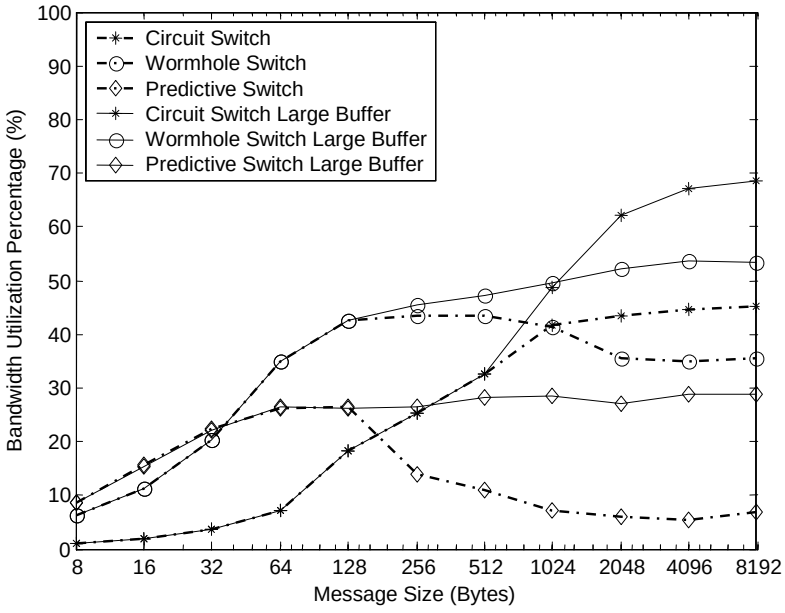


(b) 100 foot cable.

Figure 8.10: SystemC simulation vs. VHDL simulation (Random-to-all communication pattern).

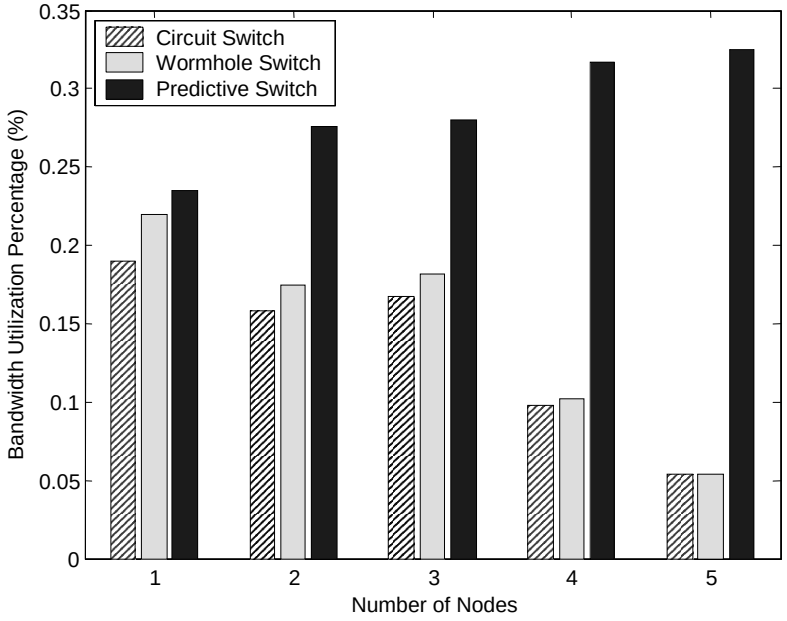


(a) 10 foot cable.

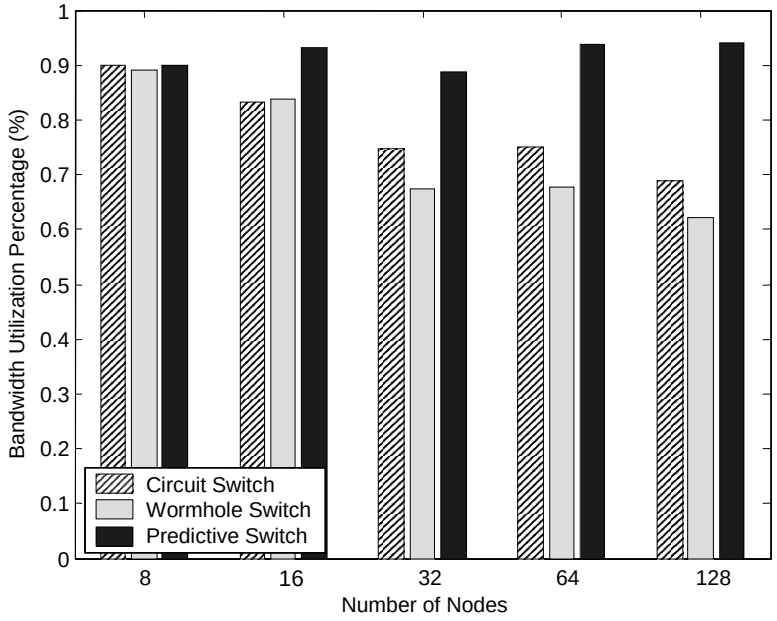


(b) 100 foot cable.

Figure 8.11: Simulation of buffer size vs. bandwidth (Random-to-all communication pattern).



(a) 16 byte message



(b) 512 byte message.

Figure 8.12: SystemC system simulation for up to 128 processors (All-to-all communication pattern, 10 foot cable).

experiments performed, we set these parameters to values that match our hardware synthesis results. Our initial experiments utilized ModelSim to execute our VHDL simulations for 32 processors. We found that scaling beyond this level exceeded the 1.5 GB memory capacity but did not fully utilized the processor. By converting to our SystemC modules and executing the software version, we validate that our two models are nearly identical and that our SystemC framework scales to 128 processors. Using SystemC, the bottleneck was processing and not memory capacity.

The frequency of the processor, *f_pe*, determines the data generation speed, and was set to 500MHz to approximate a fast processor/NIC interface. The buffer size for the N-Queue component is small at 128 bytes, which is 16 64-bit words. In our "large buffer" simulations, this is increased to 240KB or 30K words but this is only shown for SystemC as this was not possible for our VHDL simulations. The clock frequency for cable *f_cable* was set to 100MHz to emulate a 6.4 Gb/s throughput as our wire simulator actually passes 64-bit words. All tests were simulated twice with short cables and with long cables. For systems using short cables, the cable latency, *l_cable*, was set to 10 nanoseconds, while for systems using moderately long cables, the cable latency was set to 100 nanoseconds. These latencies approximate 10 foot and 100 foot cables. The scheduler's working frequency, *f_scheduler*, was set to 100MHz, with a latency, *l_scheduler*, of 2 cycles. The clock frequency of the switch fabric, *f_switch*, was also set to 100MHz with a latency, *l_switch*, of 1 cycle as this fabric does not contain any buffers and could be in the analog domain.

We show our results in terms of the effective bandwidth which we calculate by dividing the total number of data bits sent by the total time requires for a set of messages. We normalize this value by dividing it by peak bandwidth of the cable.

In our design and simulation framework, we propose that the modules be designed using a hardware design flow to extract performance characteristics. To avoid the scaling problem that is inherent to VHDL simulations, we utilize the performance characteristics of the VHDL module design to create equivalent SystemC components that are then integrated into a larger system. To verify that this methodology is accurate, we have simulated the random-to-all traffic pattern in both VHDL and in SystemC. As can be seen from the performance curves in Figure 8.10, the solid curves are generated from SystemC simulation results and the dashed

curves are generated from VHDL simulation results. The SystemC and VHDL simulations are nearly identical. We find that for small messages, predictive circuit switing and wormhole routing are close in performace. However, for long cables, the performance of circuit switching drops because the distance of the control path between the NIC and the switching node increases. Therefore, it takes more time to establish circuit connections. Because the random traffic has very low predicability, in the predictive switching technique all 32-destination configurations are preloaded for each processor. All 32-destination configurations are rotated sequentially in round-robin manner. If there is one message to be sent, it must wait for the average of 16 communciation cycles, which drastically increases the message latency. Each communication cycle can send 80 bytes, so message that requires more than one cycle, it must wait for 31 cycles before it can continue to send its data. Hence, the hit ratio is very low. This describes the worst-case scenario for predictive circuit switching. The overall performance of the predictive switching, circuit switching and wormhole routing drops considerably when the message size over 128 bytes. This is due to the limited buffer size. After the buffer is full, the scheduler within the switching node will have fewer options to route packets.

One of the benefits of using SystemC is more efficient use of memory. We tried to increase the buffer size of the N-Queue in our VHDL simulations but we ran out of physical memory. We expanded the buffer size for each destination to 240 Kbytes in SystemC simulation and noticed the expected performance improvement. Figure 8.11 shows the bandwidth utilization for both small (128 byte) and large (240 kilobyte) buffers.

To illustrate the benefit of the scaling our simulations to 128 processors, we simulated the *all-to-all* communications pattern for all three networks using ten foot cabling. When the number of nodes is small, predictive switching, circuit switching and wormhole switching have similar performance. However, when the number of nodes increases to 128 for small messages (16 bytes), predictive switching significantly outperforms the others, shown in Figure 12(a). For medium sized messages, the benefits of predictive switching become more pronounced as the system size increases, shown in Figure 12(b).

8.6 CONCLUSIONS

This chapter has presented a common framework for designing, synthesizing, and simulating parallel computing networks. By using a hardware design flow, each component can be designed separately and characterized in terms of latency and bandwidth. By using FPGAs as the target technology, we are able to present performance results that can be compared against, and give insight into, ASIC performance. The hardware synthesis tools provide a maximum frequency of the device, and from simulations we can determine the latency in terms of clock cycles. By multiplying the cycle latency and the device frequency, we can accurately determine the latency down to the nanosecond (10^{-9}) level of accuracy.

By making our framework modular, we are able to create different networks using components. The input and output files provide the network traffic. By using the VHDL hardware description language with a hardware simulator, we are able to simulate the entire network to cycle accuracy using communication traces. The network performs the actual routing and contention arbitration necessary to route data through a large parallel computing network. This level of system simulation enables us to examine the true behavior of the network with a specific set of parameters, and with specific switching techniques.

9.0 CONCLUSION AND FUTURE DIRECTIONS

This dissertation proposes a new network switch architecture, the hybrid switch, by combining the predictive circuit switch and the wormhole switch in a single switch using virtual channels.

9.1 CONCLUSION

The hybrid switch takes advantage of both wormhole switching and predictive circuit switching. Small and un-predictable traffic is transferred through wormhole switch to achieve flexibility. Predictable traffic is transferred through predictive circuit switching, therefore avoiding control and buffering overhead. The overhead of predictive circuit switching caused by connection establishment is amortized by reusing established connections in a time-division multiplexing approach. By dynamically selecting the proper switching technique based on the type of communication traffic, the hybrid switch improves effective bandwidth utilization for most types of traffic.

9.2 PRIMARY CONTRIBUTIONS

The dissertation proposed a hybrid switching technique. Scheduling and virtual channel assignment issues are investigated. A cycle-accurate simulation framework has been built to evaluate the proposed hybrid switching technique. The contributions of the dissertation are listed as below.

1. A hybrid switch architecture.

   The dissertation proposes a hybrid switch architecture combining predictive circuit switching and wormhole switching methods. Data are transferred through high-throughput and low-cost analog switch fabrics.

2. Optimizing virtual channel assignment schemes

   Three virtual channel assignment schemes are introduced, which are SES (skip empty slots), SLA (slot length adjustment) and PREEMPT (preempt slots). With the virtual channel assignment scheme, the hybrid switch successfully handles both predictable traffic and un-predictable traffic.

3. A real-time greedy scheduler.

   The predictive circuit switching scheduler facilitates the hybrid switching system. Predictive circuit connections are established before traffic comes and reused multiple times to amortize the control overhead caused by establishing circuit connections.

4. Optimizing scheduling algorithms.

   Two optimizing scheduling algorithms are described. One is an optimizing scheduling method for crossbar networks. The other is a Level-Wise scheduling algorithm for fat tree interconnection network. In predictive circuit switching, connections are kept for a long time, called long-lived connections. The optimized scheduling algorithms are able to improve the effective bandwidth utilization.

5. A uniform network simulation framework

   SystemC based network simulator provides a flexible and uniform simulation framework to evaluate our new switching technique.

9.3 FUTURE DIRECTIONS

Future directions of this research include compiler design, multiple crossbar architecture and prototype implementation.

9.3.1 Intelligent compiler

Compiled communication is the basis of the hybrid switch. An intelligent compiler will boost the network performance based on a hybrid switching. Commands should be inserted into parallel programs to indicate communication patterns, starting and ending points of a phase. As indicated in our simulation result, the rough traffic ratio computed as predictable traffic over total traffic is also helpful.

9.3.2 Hardware prototype

The dissertation proposes a hybrid switch architecture. Most dissertation research is based on architecture designs and simulations. A proof-of-concept hardware prototype targeted on FPGA evaluation board is an interesting direction. Four Xilinx Vertex evalaution boards can be stacked together with LVDS cables to build a fundamental network.

APPENDIX

SELECTED PUBLICATIONS

1. Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem, "Switch design to enable predictive multiplexed switching in multiprocessor networks," in *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, (Denver, CO), Apr. 2005.
2. R. Hoare, Z. Ding, S. Tung, A. Jones, and R. Melhem, "A framework for the design, synthesis and cycle-accurate simulation of multiprocessor networks," *Journal of Parallel and Distributed Computing (JPDC)*, 2004.
3. R. Hoare, S. Tung, B. Farren, and Z. Ding, "Incorporation of physical layer characteristics into system level modeling of large digital systems," in *Proc. Int. Conf. on Applied Modeling and Simulation, IASTED2002*, 2002.

BIBLIOGRAPHY

[1] S. F. Nugent, "The iPSC/2 direct-connect communications technology," in *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues*, vol. 1, pp. 51–60, ACM Press, 1988.

[2] W. Oed, "The Cray research massively parallel processing system: Cray:T3D." Cray Research, 1993.

[3] IBM, "IBM eserver pSeries SP switch and SP switch2 performance." http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/, February 2003.

[4] F. Petrini, W. chun Feng, A. Hoisie, and S. Chaudhury, "The Quadrics network: high-performance clustering technology," *Micro, IEEE*, vol. 22, no. 1, pp. 46 – 57, 2001.

[5] J. Shalf, S. Kamil, L. Oliker, and D. Skinner, "Analyzing ultra-scale application communication requirement for a reconfigurable hybrid interconnect," in *IEEE Conference on Supercomputing*, 2005.

[6] K.J.Barker, A. Benner, R. Hoare, A. Hoisie, A. Jones, D.J.Kerbyson, D.Li, R. Melhem, R.Rajamony, E. Schenfeld, and C.Stunkel, "On the feasibility of optical circuit switching for high performance computing system," in *IEEE conference on Supercomputing*, 2005.

[7] National Semiconductor Co., "SCAN50C400 1.25/2.5/5.0 GBPS quad multi-rate backplane transceiver." Data Sheet, Jan. 2004.

[8] J. Duato, P. Lopez, and S. Yalamanchili, "Deadlock- and livelick-free routing protocols for wave switching," in *11th International Parallel Processing Symposium*, pp. 570–577, 1997.

[9] J. Duato, P. Lopez, F. Silla, and S. Yalamanchili, "A high performance router architecture for interconnection networks," in *in Proc. 1996 Int. Conf. Parallel Processing*, vol. 1, 1996.

[10] The BlueGene/L Team, "An overview of the BlueGene/L supercomputer," in *Conf. on High Performance Networking and Computing*, 2002.

[11] PLX Technologies, "RLX ultra high desnsity balde servers." http://www.nasi.com.

[12] C. B. Stunkel, "Commercial MPP networks: Time for optics," in *4th Int. Conf. on Massively Parallel Processing Using Optical Interconnections (MPPOI'97)*, pp. 90–95, 1997.

[13] D. Addison, J. Beecroft, D. Hewson, M. McLaren, and F. Petrini, "Quadrics QsNet II: a network for supercomputing applications." http://www.hotchips.org/archive/hc15/pdf/2.quadrics.pdf.

[14] National Semiconductor Co., "National LVDS products." http://www.national.com, 2004.

[15] National Semiconductor Co., "SCAN90CP02 1.5 Gbps 2x2 LVDS crosspoint switch with pre-emphasis sand IEEE 1149.6." Data Sheet, Feb. 2004.

[16] National Semiconductor Co., "DS90CP04 4x4 low power 2.5 Gb/s LVDS digital crosspoint switch." Data Sheet, Jan. 2004.

[17] EXFO Co., "Optical switch IQ-9100." http://documents.exfo.com/specsheets/IQ-9100an.pdf, 2001. Data Sheet.

[18] J. Lee and S.-T. Ho, "Ultra-high-capacity optical communications and networking." http://oclab.usc.edu/nsf/oct2002/pdf/lee-lisos.pdf, 2002. Northwestern University.

[19] B. VanVoorst and S. Seidel, "Comparison of MPI implementations on a shared memory machine," in *in Proceedings of the 15th IPDPS 2000 Workshops on Parallel and Distributed Processing*, no. 847-854, 2000.

[20] T. Tabe and Q. F. Stout, "The use of the mpi communication library in the nas parallel benchmarks." http://citeseer.ist.psu.edu/tabe99use.html.

[21] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI the complete reference*. The MIT Press, 1996.

[22] X. Yuan, *Dynamic and Compiled Communication in Optical Time Division Multiplexing Point-to-Point Networks*. PhD thesis, University of Pittsburgh, 1998.

[23] A. Afsahi and N. Dimopoulos, "Collective communication on a reconfigurable opitcal interconnect," in *Proceedings of the OPODIS'97, International Conference on Principles of Distributed Systems*, pp. 167–181, 1997.

[24] A. Afsahi, *Design and Evaluation of Communication Latency Hiding/Reduction Techniques for Message-Passing Enviroments*. PhD thesis, University of Victoria, Canada, 2000.

[25] X. Yuan, R. Melhem, and R. Gupta, "Algorithms for supporting compiled communication," *Trans. on Parallel and Distributed Systems, IEEE*, pp. 107–118, 2003.

[26] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng, "The SUIF compiler for scalable parallel machines," in *7th SIAM Confernece on Parallel Processing for Scientific Computing*, 1995.

[27] A. Afsahi and N. J. Dimopoulos, "Hiding communication latency in reconfigurable Message-Passing enviroments," Tech. Rep. ECE-99-3, Victoria, B.C., Canada, 1999.

[28] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, "Predicting multiprocessor memory access patterns with learning models," in *Proceedings of the Fourteenth International Conference on Machine Learning*, pp. 305–312, 1997.

[29] E. Authurs, M. S. Goodman, H. Kobrinski, and M. Vecchi, "HYPASS: An optoelectronic hybrid packet switching system," *IEEE Journal on Selected Area in Communications*, vol. 6, no. 9, pp. 1500–1510, 1988.

[30] J. Livas, T. Hofmeister, and K. Horne, "Taking a hybrid optical-switch approach." http://www.eetimes.com/article/showArticle.jhtml?articleId=18310898, 2004. EE Times.

[31] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie, "Hardware- and software-based collective communication on the quadrics network," in *IEEE International Symposium on Network Computing and Applications*, Feb 2001.

[32] S. Coll, J. Duato, F. Mora, F. Petrini, and A. Hoisie, "Collective communication patterns on the quadrics network," *Performance Analysis and Grid Computing*, 2003.

[33] R. Ponnusamy, R. Thakur, A. Choudhary, and G. Fox, "Scheduling regular and irregular communication patterns on the cm-5," in *Proceedings of Supercomputing 92*, pp. 394–402, November 1992.

[34] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits, "Using multirail networks in high-performance clusters," in *Proceedings of the 2001 IEEE International Conference on Cluster Computing*, pp. 15–24, Oct. 2001.

[35] Y.-L. Chen and J.-C. Liu, "A hybrid interconnection network for integrated communication services," in *11th International Parallel Processing Symposium (IPPS '97)*, pp. 341–345, 1997.

[36] C. Salisbury and R. Melhem, "A high speed scheduler/controller for unbuffered banyan networks," in *Proceedings of the IEEE International Conference on Communications*, vol. 1, pp. 645–650, 1998.

[37] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz, "Tiny Tera: A packet switch core," *IEEE Micro*, vol. 17, no. 1, pp. 26–33, 1997.

[38] N. McKeown, *Scheduling algorithms for input-queued cell switches*. PhD thesis, University of California at Berkeley, 1995.

[39] F. M. Chiussi and A. Francini, "Scalable electronic packet switches," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 486–500, May 2003.

[40] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The QuadricsNetwork (QsNet): high-performance clusting technology," in *the 9th IEEE Hot Interconnects (HotI'01)*, 2001.

[41] A. Ganz and Y. Gao, "Tdma communication for ss/tdma satellites with optical intersatellite links," in *IEEE International Conference on Commutations*, vol. 3, pp. 1081–1085, 1990.

[42] C. Qiao and R. Melhem, "Dynamic reconfiguration of optically interconnected networks with time division multiplexing," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 268–278, 1994.

[43] K. L. Yeung, "Efficient time slot assignment algorithms for tdm hierarchical and non-hierarchical switch systems," *IEEE Transactions on Computers*, vol. 49, February 2001.

[44] R. Melhem, "Time-multiplexing optical interconnection networks; why does it pay off?," in *In Proc. of the ICPP workshop on Challenges for Parallel Processing*, 1995.

[45] J. Liang, S. Swaminathan, and R. Tessier, "ASOC: a scalable, single-chip communications architecture," in *International Proceedings on Parallel Architectures and Compilation Techniques*, pp. 37–46, Oct. 2000.

[46] A. Faraj and X. Yuan, "Communication characteristics in the nas parallel benchmarks," in *Proc. 14th IASTED Int. Conf. on Parallel and Distributed Computing and Systems, IPDCS 2002*, (Cambridge, MA), pp. 729–734, November 2002.

[47] D. Lahaut and C. Germain, "Static communications in parallel scientific programs," in *In Proc. of PARLE*, 1994.

[48] W. Thies, M. Karczmarek, and S. Amarsinghe, "StreamIt: a language for streaming applications," in *In Proc. og the Int. Cpmf. on Compiler Construction*, 2002.

[49] S. Hinrichs, *Compiler directed architecture-dependent communication optimization*. PhD thesis, Carnegie Mellon University, 1995.

[50] T. Gross, "Communication in iWarp ssytems," in *In Proc. of Supercomputing*, 1989.

[51] T. Gross, A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker, "Communication styles for parallel systems," *IEEE Computer*, 1994.

[52] F. Cappello and C. Germain, "Toward high communication performance through compiled communications on a circuit switched interconnection network," in *in Proc. of First IEEE Symposium on High-Performance Computer Architecture*, 1995.

[53] G. Viswanathan and J. Larus, "Compiler-directed sharedmemory communication for iterative parallel applications," in *in Proc. of the 1996 ACM/IEEE Conf. on Supercomputing*, 1996.

[54] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, "Predicting multiprocessor memory access patterns with learning models," in *Proc. 14th Int. Conf. on Machine Learning*, pp. 305–312, 1997.

[55] S. Kaxiras and C. Young, "Coherence communication prediction in shared-memory multiprocessors," in *in Proc. of the 16th Int. High Performance Computer Architecture*, 2000.

[56] F. Glover, "Maximum matching in a convex bipartite graph," *Naval research Logistics Quarterly*, 1967.

[57] A. L. Dulmag and N. S. Mendelsohn, "Actrices associated with the hitchcock problem," *Journal of the ACM*, 1962.

[58] Z. Galil, "Efficient algorithms for finding maximum matching in graphs," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 23–38, 1986.

[59] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani, "Matching is as easy as matrix inversion," in *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, (New York, New York, United States), pp. 345–354, 1987.

[60] S.Habata, M. Yokokawa, and S.Kitawaki, "The Earth simulator system," tech. rep., NEC Res. and Develop. http://www.owlnet.rice.edu/ elec526/handouts / papers/earth-sim-nec.pdf.

[61] T. Weller and B. Hajek, "Scheduling nonuniform traffic in a packet-switching system with small propagation delay," *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 813–823, 1997.

[62] K. L. Yeung, "Efficent time slot assignment algorithms for TDM hierarchical and non-hierarchical switch systems," *IEEE Trans. on Communications*, vol. 49, pp. 351–359, Feb. 2001.

[63] N. Mckeown, "Crossbar switch scheduling," *SNRC Talk*, 2001.

[64] L.Lovasz and M.Plummer, *Matching Theory*. Academic Press, 1986.

[65] H. N. Gabow, "An efficient implementation of edmonds' algorithm for maximum matching on graphs," *Journal of the ACM (JACM)*, vol. 23, no. 2, pp. 221 – 234, 1976.

[66] M. A. Czygrinow and E.Szymanska, "Distributed algorithm for approximating the maximum matching," *Elservier Science*, 2003.

[67] M. Hanckowiak and M. K. A. Panconesi, "On the distributed complexity of computing maximal matchings," in *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, (San Francisco, California, United States), pp. 219–225, 1998.

[68] C.E.Leiserson, "Fat trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. on Computers*, vol. 34, no. 10, pp. 892–901, 1985.

[69] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, and et al., "The network architecture of the connection machine CM-5," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pp. 272–285, 1992.

[70] Z.Bozkus, S.Ranka, and G.Fox, "Benchmarking the CM-5 multicomputer," in *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 100–107, 1992.

[71] K.E.Schauser and C.J.Scheiman, "Experience with active messages on the Meiko CS-2," in *9th International Parallel Processing Smposium*, pp. 140–149.

[72] "AlphaServer SC:terascale single-system-image supercomputing," tech. rep., COMPAQ Inspiration Technology, 2002. http://h18002.www1.hp.com/alphaserver/download/.

[73] J. Beecroft, D. Addison, F. Petrini, and M. McLaren, "QsNetII: an interconnect for supercomputing applications," *IEEE Micro*, 2003.

[74] L. M. Ni, Y. Gui, and S. Moore, "Performance evaluation of switch-based wormhole networks," *IEEE transaction on parallel and distributed ssytems*, vol. 8, no. 5, pp. 462–474, 1997.

[75] Y. Aydogan, C. B. Stunkel, C. Aykana, and B. Abali, "Adaptive source routing in multistage interconnection networks," in *10th International Parallel Processing Symposium (IPPS '96)*, pp. 258–267, 1996.

[76] H.Kariniemi and J. Nurmi, "New adaptive routing algorithm for extended generalized fat trees on-chip," in *International Symposium on System-on-Chip* (IEEE, ed.), pp. 113–118, 2003.

[77] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrstructure of computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[78] L. Breslau, D. Estrin, K. F. adn Sally Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu, "Advances in network simulation," *IEEE Computer*, vol. 33, no. 5, pp. 59–67, 2000.

[79] T. Cegrell, "A simulation model of the TIDAS computer network," *IEEE Transactions on Communications*, vol. 24, no. 3, pp. 355–358, 1976.

[80] P. Mars, "Some aspects of simulation in telecommunication networks," in *Twelfth UK Tele-traffic Symposium, Performance Engineering in Telecommunications Networks (Digest No. 1995/054)*, pp. 1/1–1/4, IEE, 1995.

[81] E. C. Russell, *Building Simulation Models with Simscript II.5*. CACI Products Company, 1999.

[82] OPNET Technologies, Inc., "Modeler: Acelerating network R&D," whitepaper, 2004.

[83] A. D. George, R. B. Fogarty, J. S. Markwell, and M. D. Miars, "An integrated simulation environment for parallel and distributed system prototyping," *Simulation*, vol. 72, pp. 283–294, May 1999.

[84] J. Rexford, W. Feng, J. Dolter, and K. G. Shin, "PP-MESS-SIM: a flexible and extensible simulator for evaluating multicomputer networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 1, 1997.

[85] Y. Liu and S. Dickey, "Simulation and analysis of enhanced switch architecture for interconnection networks in massively parallel shared memory machines," in *Proceedings of 2nd Symposium on the Frontiers of Massively Parallel Computation*, pp. 487–490, 1988.

[86] I. Gorton, J. Kerridge, and B. Jervis, "Simulating microprocessor systems using occam and a network of transporter," in *IEE Proceedings on Computers and Digital Techniques*, vol. 136, pp. 22–28, 1989.

[87] K. Wilson and J. Glodoveza, "Mentor gaphics unveil powerful synthesis tool to meet requirements of next-generation programmable logic design," tech. rep., Mentor Graphics, 2002.

[88] A. Ki, B.-I. Park, J.-G. Lee, and C.-M. Kyung, "Transaction level modeling of SoC with SystemC 2.0," in *SOC Design Conf.*, 2003.

[89] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara, "From VHDL register transfer level to systemc transaction level modeling: a comparative case study," in *Proc. 16th Int. Symposium on Intergrated Circuits and Systems Design*, 2003.

[90] Mentor Graphics, "ModelSim," tech. rep., Mentor Graphics, 2000.

[91] Mentor Graphics, "Design exploration tutorial," tech. rep., Mentor Graphics, 2001.

[92] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete data sheet," tech. rep., Xilinx Inc., 2004.

[93] Altera, "Stratix II device handbook," tech. rep., Altera Inc., 2004.

[94] R. Hoare, S. Tung, B. Farren, and Z. Ding, "Incorporation of physical layer characteristics into system level modeling of large digital systems," in *Proc. Int. Conf. on Applied Modeling and Simulation, IASTED2002*, 2002.

[95] R. Hoare, Z. Ding, S. Tung, A. Jones, and R. Melhem, "A framework for the design, synthesis and cycle-accurate simulation of multiprocessor networks," *Journal of Parallel and Distributed Computing*, 2004.

[96] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem, "Switch design to enable predictive multiplexed switching in multiprocessor networks," in *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, (Denver, CO), Apr. 2005.